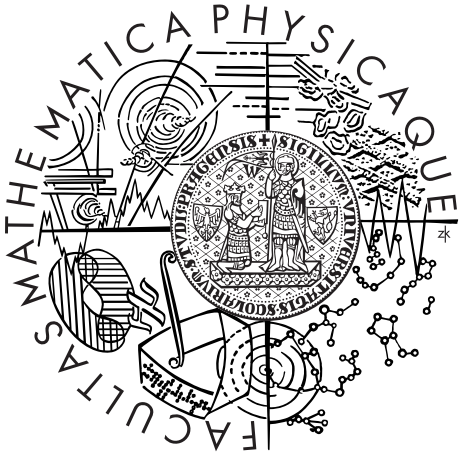
Charles University in Prague
Faculty of Mathematics and Physics

MASTER THESIS



Bc. Radek Píbil

Integrace Pogamutu s Defconem

Department of Software and Computer Science Education

Supervisor of the master thesis: Mgr. Jakub Gemrot
Study programme: Informatics
Specialization: Theoretical Computer Science

Prague 2011

I would like to thank the following list of people: Jakub Gemrot for helping me with all steps of my master thesis, by consultation on code, text and thesis' direction, Cyril Brom for the general leadership and ideas on thesis' direction as well, Robin Baumgarten for the help with the Defcon AI API and resources he provided, and also AMIS group as a whole.

Finally, I would like to thank anyone who supported me during writing my master thesis, of whom the most important is my family.

I declare that I carried out this master thesis independently, and only with the cited sources, literature and other professional sources.

I understand that my work relates to the rights and obligations under the Act No. 121/2000 Coll., the Copyright Act, as amended, in particular the fact that the Charles University in Prague has the right to conclude a license agreement on the use of this work as a school work pursuant to Section 60 paragraph 1 of the Copyright Act.

In ..... date ..... Signature

Název práce: Integrace Pogamutu s Defconem

Autor: Bc. Radek Píbil

Katedra: Kabinet software a výuky informatiky

Vedoucí diplomové práce: Mgr. Jakub Gemrot

Abstrakt: V této práci se zabýváme napojením platformy pro vývoj umělé inteligence Pogamut k PC hře Defcon. Defcon je strategie v reálném čase pro více hráčů, ve které hráč hraje za jednu ze světových velmocí a velí její námořní, letecké a jaderné síle. Naším důvodem pro napojení Pogamutu na Defcon je zvyšující se důraz na herní UI mezi akademickými tématy, jdoucí ruku v ruce s neustále větším počtem počítačových her dovolujících implementovat programátorům vlastní UI. Platforma Pogamut pro vývoj UI nedávno rozšířila podporované světy o Starcraft a Defcon. Tyto dva světy ji velmi obohatili, protože na rozdíl od původních světů nejde o hry z pohledu první osoby, ale o strategie v reálném čase. V této práci máme pět hlavních témat. Za prvé popíšeme napojení Pogamutu na Defcon. Za druhé se budeme zabývat algoritmy užitečnými pro vývoj agentů pro Defcon. Za třetí ukážeme implementaci agenta v jazyce Java a za čtvrté implementaci agenta za použití MAS architektury Jason. Práci zakončíme vyhodnocením úspěšnosti těchto agentů.

Klíčová slova: Defcon, Umělá inteligence, Strategie v reálném čase, Pogamut

Title: Bridging Pogamut and Defcon

Author: Bc. Radek Píbil

Department: Department of Software and Computer Science Education

Supervisor of the master thesis: Mgr. Jakub Gemrot

Abstract: In this thesis we are going to discuss the support of Pogamut AI framework for the Defcon PC game. Defcon is a multiplayer real-time strategy putting player into control of one part of the world's sea force, air force and nuclear arsenal. We are going to cover five main topics. First is concerned with bridging Pogamut and Defcon. Next discusses provided algorithms useful for agent programming for such a kind of environment. Third describes the implementation of a purely Java agent. Fourth shows an implementation using Jason MAS framework. Final is going to evaluate the performance of the agents. Our main reason for bridging Pogamut is that as the gaming AI becomes more and more prominent in academia, more and more computer games allow programmers to implement their own AI. Pogamut AI platform follows this trend by expanding into two new environments Starcraft and Defcon, which introduce real-time strategy environments to Pogamut, whose origins are in first-person shooters.

Keywords: Defcon, Artificial Intelligence, Real-Time Strategy Games, Pogamut

Contents

Introduction **4**
Thesis Goals . . . 6
Thesis Structure . . . 6

1 Pogamut **7**
1.1 Introduction . . . 7
1.2 Early History . . . 7
1.3 Worldview . . . 7
1.4 Layered Architecture . . . 7
1.5 Other Features and Applications . . . 8
1.6 Summary . . . 9

2 Defcon **10**
2.1 Introduction . . . 10
2.2 Game Setup . . . 10
2.3 Gameplay . . . 11
2.4 Defcon AI API . . . 12
2.5 Automatic Aspects of Defcon AI . . . 12
2.6 Issues of Defcon AI API . . . 13
2.7 Installing Defcon AI API . . . 13
2.8 Agent Loading . . . 13
2.9 Summary . . . 14

3 PogamutDefCon **15**
3.1 Introduction . . . 15
3.2 Java Native Interface . . . 15
3.3 Using JNI with Defcon . . . 15
3.4 Worldview . . . 16
3.5 Guice . . . 17
3.6 Agent . . . 17
3.7 Logic Controller . . . 17
3.8 Agent Module . . . 18
3.9 Fleets Manager . . . 18
3.10 Buildings Manager . . . 18
3.11 PogamutDefCon Threads . . . 18
3.12 GameInfo . . . 19
3.13 Running PogamutDefCon Agent . . . 19
3.14 Summary . . . 20

4 Map Analysis **21**
4.1 Introduction . . . 21
4.2 Finding Points of Interest . . . 21
4.2.1 Areas . . . 21
4.2.2 Accessing the Map Data . . . 22
4.2.3 Quad Trees over Areas for Fleet and Building Placements . 22

4.3 Summary 24

5 Multi-agent systems and PogamutDefcon 26
5.1 Introduction 26
5.2 Typical Multi-Agent System 26
5.3 Multi-Agent System for Defcon 27
5.4 Pros of MAS Paradigm 27
5.4.1 Modularity 27
5.4.2 Logic Background 27
5.4.3 Game Theory 28
5.4.4 Agent Oriented Programming 28
5.4.5 Robotics 28
5.5 Cons of MAS paradigm 28
5.5.1 Debugging Cooperative Eefforts 28
5.5.2 Higher Resource Consumption 29
5.6 Summary 29

6 Proof-of-concept AI For Defcon Using PogamutDefCon 30
6.1 Introduction 30
6.2 Main Agent 30
6.3 Silo Agent 31
6.4 Airbase Agent 31
6.5 Mixed Fleet Agent 32
6.6 Note on Resource Requirements 33
6.7 Summary 33

7 Jason 34
7.1 Introduction 34
7.2 Belief-Desire-Intention 34
7.3 AgentSpeak(L) 35
7.4 Jason 35
7.5 Agent 36
7.6 Agent Architecture 37
7.7 Internal Actions 37
7.8 Belief Base 37
7.9 Environment 38
7.10 Execution Control 38
7.11 Runtime Services 38
7.12 Running a Jason Multi-Agent System 38
7.13 Debugging 39
7.14 Summary 41

8 Jason based AI for Defcon 42
8.1 Introduction 42
8.2 Connecting Jason with PogamutDefCon 42
8.3 Jason Main Agent 43
8.4 Jason Mixed Fleet Aagent 44
8.5 Jason Building Agents 45
8.6 Note on Implementation 45

8.7 Summary 45

9 Evaluation **46**
9.1 Introduction 46
9.2 Testbed 46
9.3 Pure Java AI Evaluation 47
9.4 Jason AI Evaluation 48
9.5 Summary and Conclusion 48

10 Final Words **51**
10.1 Related Works 51
10.2 Future Work 51

Conclusion **52**

Bibliography **53**

List of Tables **56**

List of Used Acronyms **57**

Introduction

Artificial intelligence (AI) is a part of computer science with a wide variety of applications from industrial planning to computer games. It is also a very active research area with a lot of topics to study encompassing neural networks, evolutionary algorithms, computer vision, game theory, multi-agent systems, machine learning, natural language processing etc. One of the main fields of AI are intelligent virtual agents (IVA). "An agent is anything that can be viewed as perceiving its environment through sensors and acting upon that environment through effectors" [28]. The idea behind IVA is to take inspiration from human reasoning and apply it wherever human-like reasoning should work best. Gaming industry is one of the fields, where imitations of human behaviour are pursued quite often.

General AI in academia is being taught from the basic concepts like FSM (finite state machine) and researched in a form of more advanced topics mentioned in the first paragraph. Gaming industry, on the other hand, prefers to be conservative and adopts more advanced IVA approaches from academia rather slowly. The gaming industry requires the agents to be able to deal with any imaginable situation, even if that means reducing the amount of possible situations, i.e. states the agent can find itself in, to a strict minimum. Still, there are many cases, where more advanced approaches might help. Introducing possible future AI programmers to more experimental approaches in AI may help with greater adoption of these ideas in gaming industry. Therefore we believe it is highly desirable.

Pogamut [15] is a framework for virtual agents created for the purpose of education and research of IVA. The first version of Pogamut built upon the GameBots [20] plugin for Unreal Tournament (UT) and adapted it for a more recent edition Unreal Tournament 2004 (UT2004) and also provided a framework for users to develop in. In 2010 it began to expand the list of supported environments to include Unreal Development Kit (UDK) [30] based demo of Unreal Tournament 3 (UT3) [32], Unreal Tournament 3 itself, and Unreal Runtime 2 (UR2) [31]. Other environments like Starcraft [29] and generic two-dimensional grid environment are a work in progress.

Pogamut's main goal is to provide researchers and students with a unified framework for a variety of environments and shield them from the platform specific programming languages and architectures of these environments, which tend to differ vastly from each other.

No AI researcher wants to concern herself with low level infrastructure algorithms. Researcher's main interest lies elsewhere: in decision making, machine learning, evolutionary algorithms etc. Also it seeks to provide students with a framework, in which they could put to practice all the important concepts and algorithms of AI they have learnt in their classes.

Pogamut's main programming language is Java. Nevertheless, it supports other languages (mostly agent-oriented) like POSH (Parallel-rooted Ordered Slip-stack Hierarchical) [9]. Java is a much friendlier language to novice programmers as opposed to C/C++, which is fast, but requires more careful programming. Still, Java is not as popular among game developers' community as C/C++, but its friendliness makes it better suited for Pogamut's goals.

Useful features of Pogamut and ongoing support led to a surprising success in the area. It is being used as an educational tool in classrooms at many universities throughout the world and it has been used for further research endeavours as well [2][19].

Unreal Tournament is a first person shooter game (FPS) and so is UT2004, UDK, UT3 and UR2 (same perspective, but without shooting). Pogamut is therefore tested mainly on FPS games. Its design though is not specifically targeted to suit only them. Pogamut is meant to be as general as possible, while retaining high usability.

Player in an FPS game is in control of a single character and perceives the environment through its character's own "eyes" and has to navigate the environment to reach whatever goal it is after without dying, or dying as little as possible. An RTS game is vastly different as it puts a player in control of multiple units at the same time, which through cooperation have to achieve a given goal. Player usually observes the environment from the bird's eye perspective, controlling units indirectly, by setting movement targets, attack targets and triggering whatever abilities the units have.

Different game design requires different agent design. An agent for an FPS game should be able to navigate the environment, collect items, fight the enemy and concern itself with its health etc. An RTS agent on the other hand should think about the whole battlefield, both what it can and cannot perceive, create plans for attack, defence and deception and also consider the economy, which is a source of new units the agent can use in battle in most RTS games. Good strategy requires careful positioning of units, proper composition of groups of units for a given task and possibly even a contingency plan or plans in case of a failure.

Defcon [11] is an RTS game as we have already alluded. It is an apocalyptic game by Introversion Software released in 2006 that puts a player into control of one of the parties engaged in a global thermonuclear war. In 2009 Robin Baumgarten of the Imperial College in London released, in cooperation with Introversion Software, an API [12] for programming of agents for Defcon thus giving the AI community another environment for intelligent agents, especially those taking the advantage of the multi-agent system paradigm.

RTS games pose problems similar to those studied by multi-agent systems (MAS). MAS include a wealth of knowledge on the subject and good practice. Pogamut and Defcon could provide another of still relatively few frameworks for commercial RTS environments that are designed with the MAS paradigm in mind. We believe that if we provide good tools for development of MAS agents for Defcon, then we may bring the industry and academia slightly closer. This would be achieved by students studying competitive IVAs targeted for a commercial environment like Defcon on the one side. While on the research side, researchers could explore more complex approaches to IVAs that are not studied in industry often. They do not need to concentrate only on more winning IVAs either, as the industry is more interested in more entertaining competitive IVAs.

Pogamut for UT2004 has many features specifically aimed at FPS games like weapon selection simplification, waypoint based path finding etc. Most of these features are useless for a Defcon agent. Still, the central features of Pogamut, that are going to be described in this thesis as well, are as useful in an RTS game as it is in an FPS game.

We believe that connecting Pogamut and Defcon will be useful, because of all the above mentioned reasons.

We would also like to extend this support even further. Jason is a MAS platform, and is very popular in academia. It builds upon even more popular language AgentSpeak(L), which is based upon the Belief-Desire-Interface (BDI) architecture. We decided it to connect to Pogamut for the use of Defcon as well.

Thesis Goals

The first objective is to allow Pogamut users create agents for Defcon. All features useful for Defcon present in Pogamut should either stay intact or be adapted in order to keep them as familiar as possible to users that have used Pogamut before.

The second objective is to provide some specialized algorithms for Defcon that would simplify agent's reasoning. The reason for them is to make sure that the user does not have to program the most frequently used infrastructure algorithms herself. Furthermore, many of these algorithms are not easy to implement and therefore tend to shift programmer's focus away from the decision making, which is the core of every intelligent agent.

The third objective is to showcase the above by implementing a Defcon agent using a set agents based upon the multi-agent system paradigm that make use of these features and also to serve as a good example for those interested in making agents for Defcon with Pogamut.

The fourth objective is the same as the third, but with use of Jason multi-agent system platform.

The last objective is evaluation of both of these implementation, to determine their competitive value.

Thesis Structure

The first chapter describes of Pogamut framework. The second chapter follows with discussion of Defcon, including an analysis of the gameplay mechanics and the provided AI API (application programming interface). In the third chapter we will perform an an in-depth analysis of Pogamut infrastructure exclusive to Defcon. The fourth chapter concentrates on listing and analysis of the infrastructure and proof-of-concept algorithms included with Pogamut for Defcon. The fifth chapter introduces a multi-agent systems view for Defcon. The sixth chapter discusses the pure Java implementation of a set of agent's playing Defcon together. The seventh chapter examines Jason multi-agent system framework. The eighth chapter analyzes our second set of agents, which uses Jason as a part of its implementation. In the ninth chapter we will evaluate both implementations and then we will conclude the thesis.

1. Pogamut

1.1 Introduction

In the introduction part of this thesis we have briefly discussed the history of Pogamut and its present state. In this chapter we are going to examine in a bit more deeply.

1.2 Early History

Original GameBots [20] served as a basis for Pogamut [15], because they allowed users to connect their own agent to Unreal Tournament (UT) environment and control their avatar (agent's body). The connection itself used sockets to allow access to them through a network interface. This allowed agents to be present in the same game regardless of the location of their execution. The socket-based approach carried on into GameBots2004 for Unreal Tournament 2004 (UT2004) and is still the main means of connection Pogamut uses.

Later, AMIS group was created, with one of its goals being maintenance and improvement of Pogamut's functionality.

1.3 Worldview

Worldview is a form of database that stores information about perceived objects and game events. It is directly inspired by knowledge representation proposals in [28], specifically by its definition of event and object as a generalized event. It is a concept similar to blackboard. Blackboard in the context of agents means a simple centralized database through which agent's modules can communicate or simply expose their important information. It allows programmers to replace a complex web of communication interfaces between modules with a much more simplified and straightforward architecture containing fewer connections between modules inherent to this star-like approach. Worldview emphasizes the database aspect, which it expands upon by a complex event-based system of reporting. A programmer can then simply listen to any event she wants to, possibly selecting only those related to a single object or class of objects. It can inform the agent about appearance and disappearance of objects from the agent's view and game-play events like a start of a game or an escalation to a next stage of a game. Worldview is so general in its concept that agents for most environments would find it useful, because all intelligent agents should base their reactions on the changes of the environment.

1.4 Layered Architecture

Pogamut is designed in a form of layers and worldview is one of them. Figure 1.1 shows a diagram of layers of the standard PogamutUT2004. The lowest layers take care of sending messages to and receiving messages from the environment

and unify the interface for worldview residing over the layers. Some of the events also represent objects new, updated, destroyed, appeared for the first time, or lost from sight. Worldview keeps track of these objects and any changes to them represented by events streaming in from the environment. It then reacts by generating more appropriate events of its own reporting these changes to the agent.

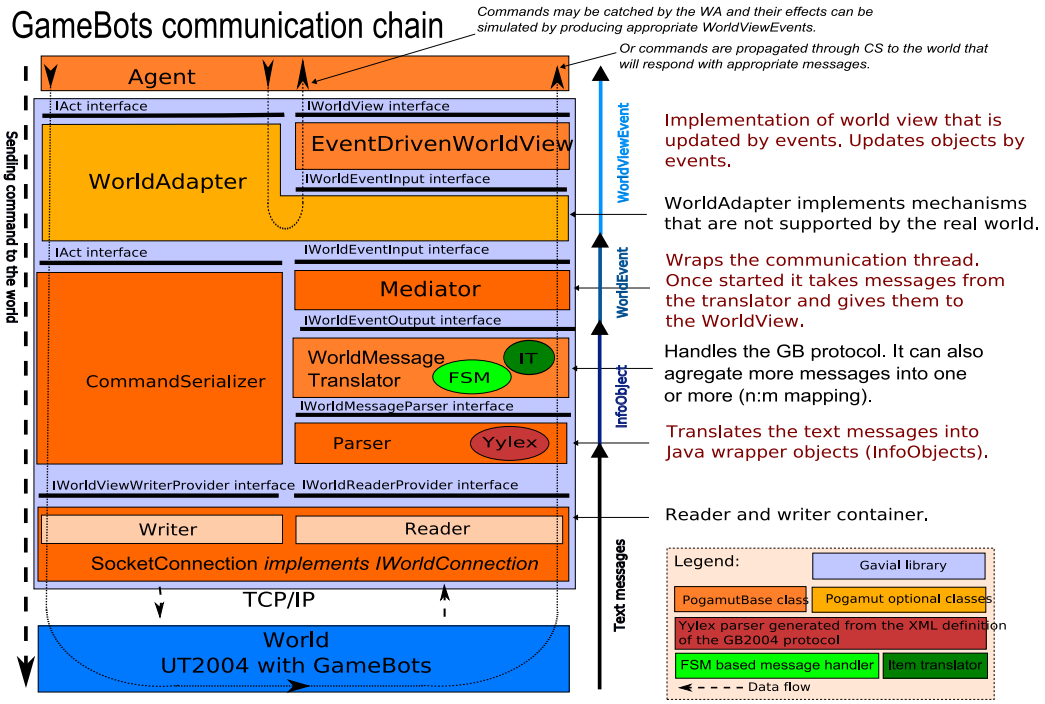


Figure 1.1: Layers of PogamutUT2004 *Source: AMIS Group*

It is expected from a programmer, who connects a new environment to Pogamut, to change the lower layers appropriately for the needs of the connection itself. The socket-based approach is thus entirely optional and can be replaced with other communication approaches. We will discuss that later in the thesis in chapter 3.

1.5 Other Features and Applications

Pogamut currently supports a wide variety of other features. Pogamut offers a connection to ACT-R, a biologically plausible cognitive architecture, by using its Java implementation jACT-R. There is also Project Emohawk [25], which is a virtual storytelling application using Pogamut. Pogamut is also a recommended framework for the BotPrize [6] competition in programming of human-like agents for Unreal Tournament 2004. Evolutionary experiment using GeneticBots [19], as mentioned in Introduction to this thesis, was based on Pogamut as well.

1.6 Summary

In this chapter, we have discussed Pogamut, its history, architecture and main feature. We have shown that layered structure allows programmer to modify each of the layers for his own needs and that worldview should be flexible enough to satisfy our needs for Defcon. Finally, we have mentioned some of other features and applications of Pogamut.

2. Defcon

2.1 Introduction

Defcon [11] is an RTS game as we have already alluded. It is an apocalyptic game by Introversion Software released in 2006 that puts a player into control of one of the parties engaged in a global thermonuclear war. We are going to explore Defcon's gameplay and the AI API (application programming interface). We have to understand Defcon's concepts in order to design agents appropriate for these conditions.

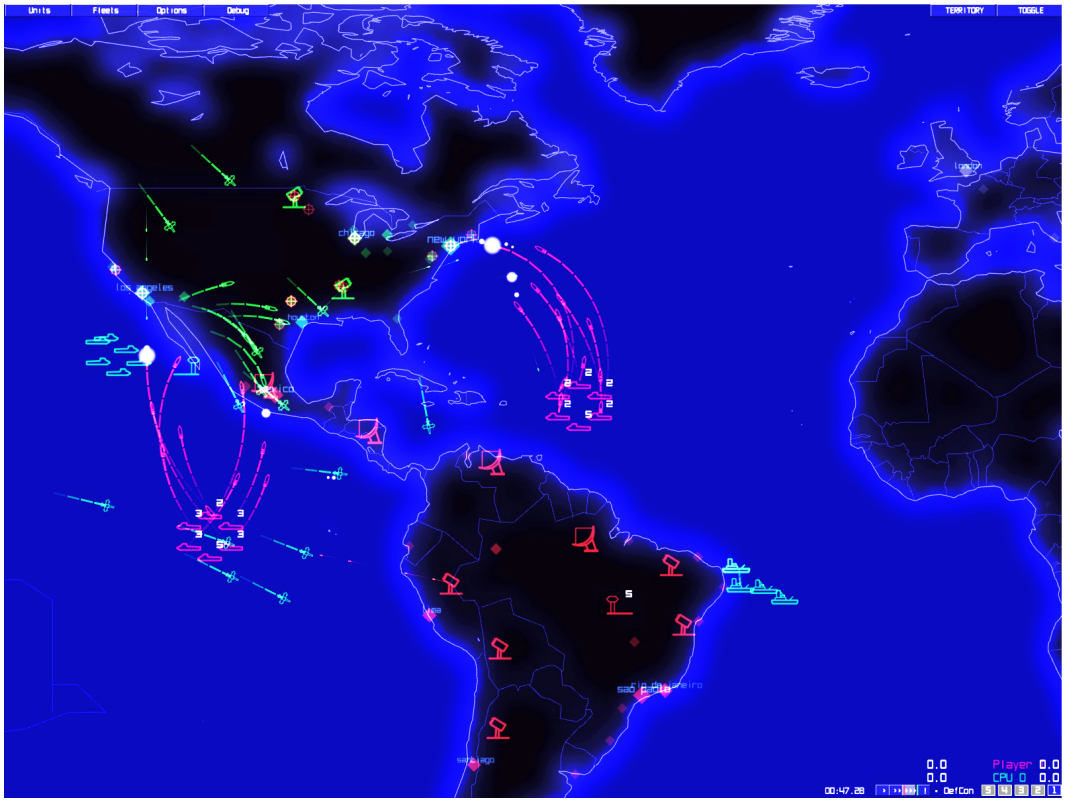


Figure 2.1: Defcon in action *Source: Introversion Sotfware, LLC*

2.2 Game Setup

Defcon is played in a flat environment with a simplified look. The default map for Defcon is a recreation of Earth rendered as a cylindrical map. The east and west are connected at the -180 and the 180 longitude, but north and south are not. Latitude coordinates under -90 or over 90 are therefore meaningless. The map consists of two types of terrain: sea and land. The list of territories consists of i) North America, ii) South America, iii) Europe, iv) Russia, v) South Asia, and vi) Africa. Each of them has a region or regions of land and sea, where the territories' owner can place his buildings and units. Spread over these land parts territories are cities, whose inhabitants are the main target of its owners' enemies.

2.3 Gameplay

Using the default game settings each type of building and unit is supplied in a fixed number. Buildings a player can choose from are i) radar, which spots incoming units including nuclear missiles ii) airbase, which hosts fighters, bombers and short range nuclear missiles for bombers iii) nuclear missile silos, which can launch nuclear missiles and also defend player's territory against incoming nuclear missiles and aircraft, while in the default anti-air defence mode.

The basic hint for placing radars is to put them very close to the parts of borders, which tend to be attacked often, but not too close to keep them defensible. Defensibility of radars is an extremely important factor in choosing their placement location, because in order to attack incoming aircrafts and nuclear missiles effectively, player's own aircrafts and silos in defensive mode require much greater vision, than any building or unit other than radar provides. Airbases are best placed more inland, because they are very fragile, and range of their aircrafts is sufficient enough in most cases. Nuclear missile silos have to be widespread all over the land part of player's territory, so they can protect all cities in their air defence mode and they also should provide sufficient variety to trajectories of nuclear missiles in their attack mode.

Placeable naval units are i) battleships ii) carriers and iii) submarines, which form fleets of up to six units. Battleships are excellent against aircraft and naval units with the exception of submarines. Carriers carry fighters, bombers and short range nuclear missiles like airbases do, but with lesser number of bombers. Carriers are perfect for strikes on enemy land territory, and they can also switch to a sonar sweep mode, which actively searches for enemy submarines. Submarines are perfect for deep strikes into enemy territory using their medium range nuclear missiles and they cannot be spotted, unless they use active sonar or nuclear missiles and also stay away from carriers and other submarines in a sonar mode. Submarines should stay hidden for most of the time of their presence on the battlefield and should try and sneak close to enemy land, without being found by enemy patrols, so they can safely unleash their considerable amounts of short range nuclear missiles.

Most buildings and units have multiple states to switch in-between, which usually take some predefined amount of time to be put to effect. The missile silos are a perfect example of a building that has multiple states. The first is their nuclear missile launch mode and the second is the anti-air defence. To change a silo's state to the nuclear missile launch state from the anti-air defence state takes 120 ingame seconds, while changing back to the anti-air state takes 340 ingame seconds.

All buildings and units have to be placed before Defcon 3. Defcon is a numerical variable that keeps track of game progress. It can have a value between 1 and 5. Defcon begins at 5 and decrements with time. Defcon 1 means that players can use nuclear weaponry. After a sufficient percent of nuclear missiles have been launch, the victory timer is triggered and the game ends, when it runs out. At the end of the game, players are evaluated based on their points. The one with the highest amount of points wins. Awarding of points is defined by the game mode. The basic mode awards two points to the owner of the nuclear missile for each victim in target city, while it subtracts one point for each of player's own

citizen lost.

The last important aspect of Defcon is alliances. The concept is very similar to alliances in table-top games like Risk or Diplomacy. Alliances do not have to be permanent and are not necessarily forged with good will on all sides. Players can consciously decide to betray their supposed allies to achieve a better score themselves. This leads to a very interesting aspect of gameplay, which should be very interesting point for studies of multi-agent systems and game theory (more on those in chapter 5).

This concludes the gameplay basics of Defcon. The next part is going to describe the Defcon AI API.

2.4 Defcon AI API

As already discussed in the Introduction to this thesis, Robin Baumgarten of the Imperial College in London released, in cooperation with Introversion Software, an API for programming of agents for Defcon in 2009 [12]. This API uses a modified executable, which adds a new option to the new game lobby to select an appropriate agent and also provides numerous command-line parameters to simplify the game setup.

Programmer is meant to implement his agent as a DLL (dynamic-link library), using a simple interface with the game executable. This interface allows a two way communication between Defcon and the DLL. The *initialise* method is meant to be used to initialize the agent. The periodical calls (usually once in $100ms$) of the *update* method let the agent to choose its next sequence of actions. Defcon also reports events that took place between updates through the *addEvent* method. These events include unit destruction, nuclear missile launch etc. The *addEvent* method is not meant to be used for the deliberation, since Defcon expects it to end as soon as possible.

The API supplies a wide variety of methods for querying Defcon and methods for issuing of commands as well. These methods provide a wide range of information about the game like whether a given location lays inside team's territory, to which team the given unit belongs, computation of the shortest path etc.

2.5 Automatic Aspects of Defcon AI

Surprisingly, Defcon itself implements some automatic behaviour for player's units. This behaviour has to be mentioned in order to be separated from AI implemented by a programmer. One example of this automatic behaviour is carriers launching their fighters or bombers immediately after an enemy unit, attackable by either one of them, appears in range of the aircraft. All units and silos in the air defence mode fire on incoming units automatically as well. The rest rests entirely on the programmer.

All of this behaviour is true even for an ordinary human player.

2.6 Issues of Defcon AI API

Defcon is available for Windows, Linux and Mac. Defcon AI API is not. It is only available for Windows. We therefore assumed Windows to be a single platform for our project.

Defcon AI API provides access to internal pathfinding through a sailing distance measuring method and through the method for issuing of movement orders. We use these methods for fleets exclusively, as neither the nuclear missiles, nor the airplanes have any meaning for them. Airplanes need only straight line distance and nuclear missiles cannot be redirected. These methods work perfectly most of the time, but sometimes, even if the location coordinates are properly formed to represent actual locations with the bounds of coordinates, the returned path is badly computed. This happens occasionally, when a movement order is issued over the map seam. The returned path happens to be wrong if the source location is close to the right side of the seam (slightly more than -180) and the target location on the left side (slightly less than 180). The path taken by the fleet takes it all around the map, and makes it close on the target location from the opposite direction. The correct path should be shorter and more direct, as the seam connects the -180 latitude with the 180 latitude. It is very difficult to realize that this bug has happened from programmer's view, since Defcon does not return the path itself, and it only accepts the target location as a movement target for the given unit. Still, this bug is rather rare, and does not come up very often.

Next issue is the buggy multiplayer game creation, if two or more players are using Defcon AI API. Once a player joins, she cannot select her own AI to play for her, because the list of AIs is empty or the joining player outright crashes during connecting to server. This left us with only humans and internal Defcon AI to compete against.

2.7 Installing Defcon AI API

The API can be found on the enclosed CD. It is one of the parts that are installed after using *PogamutDefCon.exe* installer into subdirectory DefConJavaInterface of the installation directory. The API requires a modified executable *defcon.exe* to replace the original one and placing the data directory inside the Defcon root directory as well. It is important to run Defcon at least once so the authentication key is initialized. If that is has not been done, and any form automatic agent loading is used, Defcon will be rendered unusable during this execution.

2.8 Agent Loading

The basic process for agent's loading consists of following steps:

1. Compilation of the DLL
2. Putting the compiled DLL along with all other necessary files into a directory inside the AI subdirectory of Defcon

3. Running the game either hosting or joining a game and selecting agent's DLL from the drop down menu under players list

For a Pogamut agent the process is slightly different and more automated and will be explained in the next chapter.

2.9 Summary

In this chapter we have explored Defcon. We have discussed Defcon game setup that is how the environment looks and how is it divided into territories. We have examined gameplay. We have shown what kind of units and buildings Defcon, how they behave, what are their strengths and weaknesses, at what time they can be placed etc. Also we have described the relation between Defcon level and options for what player can do at the given time.

We have explained Defcon AI API and how does it reflect on the implementation of AI. We have mentioned the automatic aspects of Defcon unit behaviour to differentiate programmer's AI's behaviour and automatic behaviour that is provided even for standard players. We have looked into issues of Defcon AI API, what issues it has, how to install it, and how to load an AI DLL.

3. PogamutDefCon

3.1 Introduction

In this chapter, we are going to analyze the bridge between Defcon and the Defcon specific implementation of Pogamut that we call PogamutDefCon.

The bridge refers to the code that connects the DLL with the Java code of Pogamut. We have selected not to use the standard Pogamut method of sockets, because Defcon itself requires a single instance for each of the AI agents. Therefore a single server with a list of attached agents (apart from the internal agents of Defcon), as is the case with Unreal Tournament, is not an option.

As a result of this realization, we chose JNI (Java Native Interface) in order to move from C/C++ code used by Defcon to Java used by Pogamut.

3.2 Java Native Interface

JNI provides a framework allowing the programmer to load the Java Virtual Machine (JVM) in a form of a DLL (another one) and call specific C/C++ methods from Java and specific Java methods from C/C++. JNI and Java call these methods native. JNI itself requires the same parameters as the executable version of Java, like the jar file. Using these parameters, it then runs the JVM, which executes the supplied code.

JNI is not as popular among programmers as one would expect, because of numerous reasons. It can be easily destabilized by subtle errors in the native code (C/C++ in our case) resulting in unpredictable crashes that are difficult to reproduce and thus are very difficult to debug. Java application also ceases to be portable, which is another major issue. Unicode string representation is also different from the standard UTF-8 and application's handling of Java strings has to be very careful and use appropriate methods provided by JVM in case of retrieving strings from its memory and their disposal as well.

However with a careful programming and thorough debugging it does function well and we also do not require portability with Defcon, because although Defcon is available for Mac and Linux, the API is not. The API is solely available for Windows, so portability is no longer possible.

3.3 Using JNI with Defcon

One of the examples for Defcon API includes a simple JNI bridge to Java. This bridge is not robust enough for Pogamut though, because there are still many pitfalls that we are going to discuss later in this chapter.

The C/C++ methods that respond to calls from Java are relatively simple. They unwrap the parameters from JVM, pass them through Defcon API, collect the data and then wrap them inside JVM types, pass them into its memory area and return back to JVM. Still, we believe that much of the earlier instability during development was caused by incorrect handling of JVM variables accessed through JNI, although we are not entirely certain, as debugging of JNI is still

an issue, as we have already stated. Calling Java methods requires more effort. Method *initialise* initializes JVM and performs the required pre-processing of parameters, like the class path, which is a list of libraries required by the agent's code. The other methods take care of incoming events from the game (*addEvent*) and agent's logic updates (*update*) during which agent can send commands to Defcon and also query it for information about game state.

Pogamut side of the bridge is the *JBot* class, through which all calls to and from Pogamut have to pass. This class passes creation of the agent to *PogamutJBotSupport* class, which collects events from Defcon, commands from the agent and also instantiates the agent itself.

Defcon AI API is not thread-safe. If methods from the API are accessed from other threads than the one, which updates the agent through the *update* method, then it might crash unexpectedly. The agent itself runs inside at least one different thread (the main logic thread) and often more. In order to solve this problem we have to buffer the queries and commands inside *PogamutJBotSupport*. When Defcon calls the *update* method new information about the world is collected, commands (maximum of 20) are performed and queries are processed up until the thread has spent more than $75ms$ inside the method. We chose $75ms$ to make sure, that the standard $100ms$ quota per update is upheld. If the quota is exceeded by a smaller indeterminate amount nothing happens, but by $500ms$ and more there is a risk of Defcon claiming that the agent has disconnected. The queries themselves are wrapped up in *SyncMethodExecContainer*, which uses reflection to store method to be executed, and keeps parameters within an array. *SyncMethodExecContainer* then uses a *CountDownLatch* to synchronize both threads appropriately.

These have been the main issues with the bridge itself and the rest of Pogamut for Defcon had to be adjusted to reflect them.

3.4 Worldview

Most of the work after reaching *PogamutJBotSupport* is performed by a slightly adapted worldview, which takes into account the thread structure of the implementation. It uses the message producer layer of the standard Pogamut layered structure to retrieve appropriate events for it to process. The message producer collects the cached info from *PogamutJBotSupport* and passes it back.

Defcon worldview and Pogamut worldviews in general can only work with events and objects of a certain type, so it was necessary to provide them. For this reason, we use a standard Pogamut approach of defining them as XML files, which are then compiled to appropriate Java classes. We have also extended this mechanism to match lower level integer constants from *JBot* that represent unit types, states and events, with user level Java enumerations, which are much more understandable.

Factory is a concept used in case an object needs to instantiate a class of an unknown object of which it has only raw data. It was necessary to supply a factory implementation for the aforementioned events and objects so *PogamutJBotSupport* does have to reference almost none of them (one exception being a fleet) outside their appropriate interface. The whole reason for this was to provide as much flexibility as possible with regards to modifications to gameplay Defcon

supports.

3.5 Guice

The following three sections one important aspect in common, they are all tied up through the Guice framework by Google [16]. The main feature of Guice is that it allows binding of appropriate classes to appropriate interfaces. As a result, programmer can ask Guice to instantiate an appropriate class knowing only an interface it expects. Guice instantiates it and also calls the correct constructor with correct parameters.

During 2010, Pogamut relied on Guice heavily, because it seemed promising that it would simplify the agent architecture therefore making programmers life easier. The result was a slight case of over designing, so it was toned down recently. Nevertheless, an ordinary programmer does not have to know much about Guice in order to use any specialized form of Pogamut.

3.6 Agent

This is the parent class of every main infrastructure class of every PogamutDefcon agent. Programmer usually does not come into contact with it very often. It is meant to be a background object and its place in the architecture comes from the standard Pogamut architecture, which is not a topic that we have or will discuss in extreme depth apart from what we have already said in chapter 1.

3.7 Logic Controller

The main class of every PogamutDefcon agent inherits *DefConAgentLogicController*. It is here, where programmer puts the all the agent's logic into. The most important methods of interest for overriding are the logic methods: *preGameLogic*, *logic*, and *firstGameLogic*.

The first one, *preGameLogic*, is being called while the game is still at lobby. During this time, most of the information about the environment is not accessible and most methods from the Defcon AI API return bogus data. This moment is useful for initialization of data structures and other objects that do not depend on them.

The second one, *logic*, is the main logic method. It gets called periodically throughout the game time. This is the place for a programmer to implement agent's deliberation.

The third one, *firstGameLogic*, is called only once after the game starts. We have found it to be the case that virtually all agents have some things to do at this moment and do not require doing them at any point later on. These can be objects that analyze map, set up the starting point of agent's deliberation with data that is not accessible during *preGameLogic* etc. Usually programmer would just add a new boolean variable to make sure that she catches the first call of the *logic* method, and then flips the variables value to avoid entering the same part of code again. We use a bit more advanced method. Java is capable of reflection,

and so we use it to tell which method is currently the appropriate logic method. It helps to avoid unnecessary conditional statements.

Another feature programmer can use is registration of his own listeners for a synchronous update before the update of the main agent's logic and after the update of worldview. We use it in a couple of instances and found it extremely useful. It simplifies updates of other objects without cluttering the main agent's code any further.

DefConAgentLogicController also hosts a mailbox for the use of objects of a type *IUnitAI*, which are going to be discussed in chapter 5 as it is an important concept for multi-agent systems.

Finally, *DefConAgentLogicController* hosts the *flagChecker*, which is used to access information about the map as we are going to discuss in chapter 4. It is also necessary for the programmer to actually assign the *flagChecker* herself. The reason for making *flagChecker* a mandatory field to be assigned manually is that many of the infrastructure objects require it in order to work properly or at all.

3.8 Agent Module

This is the class that implements a module for Guice, by registering all the appropriate classes for it. Children of this class hold both the selected child classes of *DefConAgent* and *DefConAgentLogicController*. Subclasses of *DefConAgentModule* and *DefConAgentLogicController* are the most important classes for programmer to implement.

3.9 Fleets Manager

In order to simplify handling of fleets, we have provided a manager that uses the worldview and listens for newly spotted and destroyed fleets and catalogues them based on their team. Own fleets are a special case. *FleetsManager* takes care of placing of player's own fleets, providing a programmer with a callback that reports on the success or failure, while also giving her an option to combine the request for placement with an initialization object, which is also passed back through the callback. Furthermore, *FleetsManager* updates all of programmer's own fleets through the logic listener it has registered at the logic controller. We discuss the use of *FleetsManager* much more deeply in chapter 6.

3.10 Buildings Manager

The analogue of *FleetsManager* for buildings is *BuildingsManager*. It does all of the tasks *FleetsManager* does, but for buildings.

3.11 PogamutDefCon Threads

We believe that in order to understand what threads work with what objects in PogamutDefcon, it is a good idea to provide a diagram. See figure 3.1.

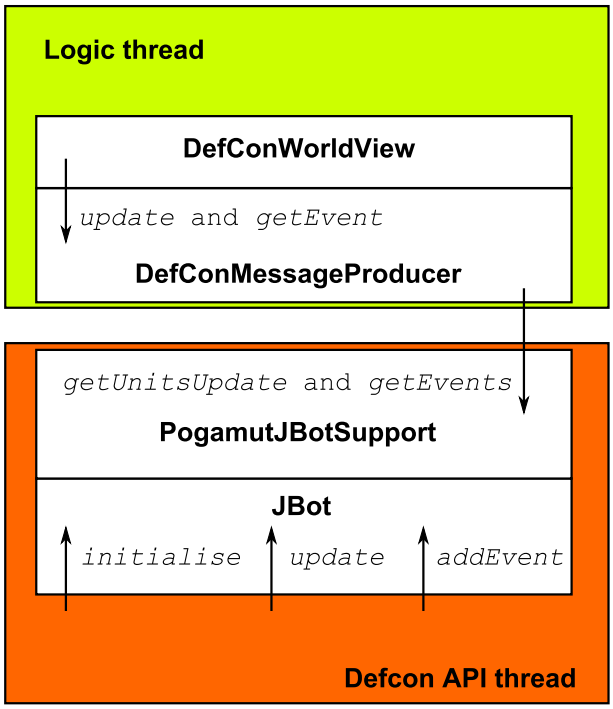


Figure 3.1: PogamutDefCon threads - a rough outline

3.12 GameInfo

Aside from worldview, user can query Defcon through a special object called *GameInfo* accessible through worldview. This object provides her with an array of methods, which either return cached information from the beginning of the game, like which territory is whose, list of teams and such, or a synchronized access to *JBot* querying methods. The list of methods is limited by those that actually query Defcon for some information and avoids exposing any of the command methods.

GameInfo uses the *SyncMethodExecContainer* (see 3.3) for each of the querying methods. If we had only one logic thread, we would be able to bypass queue and post only a single query, but, as we will discuss in the MAS Perspective chapter 6, we cannot expect that to hold in most implementations.

It is also very important to say that given *GameInfo*'s nature, programmer cannot expect returned values, which have used Defcon API and not a cached value, to be synchronous with the latest worldview update.

3.13 Running PogamutDefCon Agent

In order to run a PogamutDefCon agent we use special Ant buildfile created specifically for this reason. It handles everything from compilation to execution. User's buildfile is meant to be a copy of one of the examples provided with PogamutDefCon.

Before running the agent it is important to set a proper module class in *DefconBot.properties* file and path to Defcon in *PogamutDefCon.properties*. Once that is done, programmer has to make sure that she has the appropriate class path

set. Class path is either set through NetBeans [22] (if the programmer chooses to use this IDE) or manually through *project.properties* file in nbproject directory of the project. It must include *PogamutDefCon.jar*, *PogamutCore.jar* and all other necessary libraries. The buildfile then takes care for copying of the libraries, preparing of parameters for Defcon and launching of Defcon itself, while loading the proper DLL and user's agent as a default.

3.14 Summary

In this chapter we have looked into PogamutDefCon itself. We have described Java Native Interface (JNI), how does it relate to Defcon AI API and Pogamut, and how important is calling methods of the API from the main thread. Then we have looked at the most important classes of PogamutDefCon. We have begun at the specialized Defcon worldview, and then we have followed up with classes for the agent itself and then we have discussed useful managers continuing with the very important *GameInfo* and finally we have explained the launching process of PogamutDefCon agent simplifying the development as much as possible.

4. Map Analysis

4.1 Introduction

Defcon is an RTS game and those require a deep analysis of the map they are played upon. One thing is receiving information about each unit's state, location etc. Another is inferring information from the map. Some of the important questions about the map are:

- How do I find a good placement for my units and buildings?
- How do I find a good target for them?
- How do they get there efficiently?
- What terrain is easily defensible?

It is rarely as easy as looking up and attacking the closest enemy of each of player's own units. Players have to take a proactive approach trying and exploiting weaknesses in enemy's defence in order to reach a comfortable position from which to achieve victory. These weaknesses can also come from the map itself, not simply from enemy's mistake.

4.2 Finding Points of Interest

4.2.1 Areas

Defcon map is divided into several areas of several different kinds. Some areas are accessible by water, some allow placement of player's own buildings, some allow placement of enemy's fleets. The original, default Defcon agent uses a precomputed set of points where to place its own units, thus it does not need any further map analysis. This, however, leads to an inflexible result, which does not consider relative position to the agent's enemy too profoundly. We have decided to mitigate this drawback by calculating these points of interest ourselves.

First, it is important to find these areas and index them. In PogamutDefCon we perform this task inside *GameMapInfoPolygons* by finding a border of such areas and then calculating a reasonable approximation of such area. Finding a list of borders for all of the areas of one type can be described using a following list of steps:

1. Treat the map as a grid-based (Defcon allows floating point locations)..
2. Prepare a bit array of visited points.
3. Check the map step by step and find a first point of an area of the given type that has not been visited yet.

4. Follow the border until the start point is reached again and during the process label all visited points as such and put them into currently constructed border's list of points. While following the border, use only the basic four directions (up, left, down, right). This has to be done in order to avoid categorizing "checker board"-like areas as a single area.

5. Continue with 3 until the last point has been reached

6. Take each border and try and straighten all borders by using diagonal directions, which usually greatly reduce the number of vertices.

Once the process is finished, we have a list of borders of all areas for each of the given types.

4.2.2 Accessing the Map Data

Defcon API provides user with an interface to access these points' types. We have implemented the area finding algorithm over the standard interface in *NativeMapSource*, but because of the results we have moved to a bitmap based map data (*BitmapMapSource*). The problem is that the interface is rather slow and therefore not ideal for massive load of queries as it is in this case. Both of them implement the *IFlagChecker* interface, which specifies all of the methods that are expected from either map source, and any other map source, programmer may wish to implement.

We have received bitmaps describing each of the areas from Robin Baumgarten and used them for this purpose. As we have found later on, they do not represent the areas perfectly. This is especially true of placeable areas. As a result, we have modified the bitmaps slightly to avoid the problematic parts. Surprisingly we have found even later on, that the internal Defcon path finding algorithm sometimes finds a finitely long distance (considers it reachable) to a point, which is inaccessible. The inaccessibility issue was revealed, when we tried to set a movement target to that point for a unit that resides precisely at the starting passed to the path finding algorithm.

After evaluating the efficiency of both approaches we have still decided to give user an option to load a list of previously computed borders for all areas of all types. Remember that the areas itself do not determine the selection of locations for unit placements and such. That is still have to be performed by later processing, which we are going to describe in the next section.

4.2.3 Quad Trees over Areas for Fleet and Building Placements

Quad tree is a well known data structure (see figure 4.1), which has all internal nodes of degree of four. It is usually used to represent and partition a square two dimensional space into four equally sized square subspaces. The data added to quad tree is contained in the external nodes. Each of the subspaces, including the top level one, has a limit on how many units of data it can contain. When this limit is about to be violated, the subspace (internal node) is partitioned and a new set of four node is created.

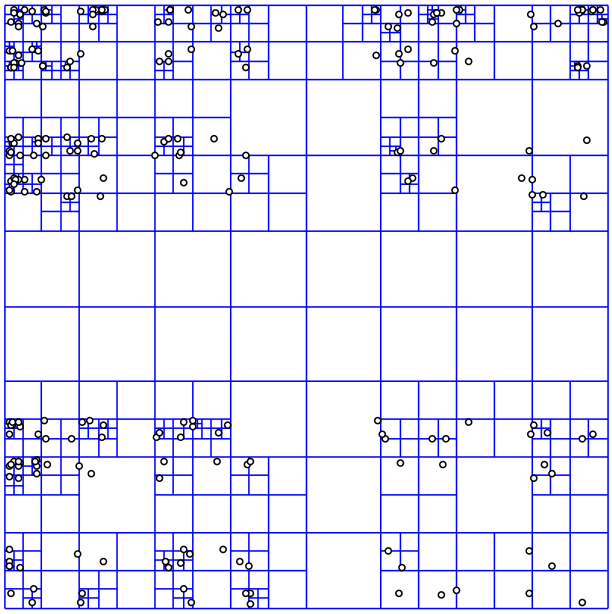


Figure 4.1: Visual quadtree example *Source: Wikipedia*

Quad trees are useful in our case, because they make it much easier to find points of interest close enough to the border, while making sure, the points are sufficiently distant from them. To clarify this statement, let's examine a problem of placing fleets.

Fleets in Defcon are groups of ships that move together. Player can place a fleet to any location, as long as all of the fleet's members are inside the naval placement area of the player and distant enough from other placed fleets. Fleet's members are offset from the placement location by given relative vectors based on the size of the fleet. Our approach to finding the best placement spot uses centre locations of enemy's naval placement and quad trees of player's own naval placement territory. We begin by calculating the diameter of the fleet and from the diameter a proper maximum depth inside the quad tree specified by 4.2 and illustrated by figure 4.3. If we cannot find an external node until this depth, we know for certain we will not be able to place the fleet anywhere inside the node, because it is too small to contain it. For each of the external nodes with lesser depth than maximum, we calculate the sailing distance through Defcon API (performs path finding) and find closest N locations to the centre of enemy area. Once we have these locations, we remember the centre of enemy area for each set to give the placed fleet a hint as to where it is supposed to go.

$$max\_depth = ceiling(\log_2 \frac{side\_of\_root\_node\_subspace}{minimum\_width})$$

Figure 4.2: Formula for calculating the maximum depth in quad tree and minimum width

This process simplified the placement selection with a reasonable approximation without a need to test every single candidate location, which is at least fleet's radius far from a single point on the border.

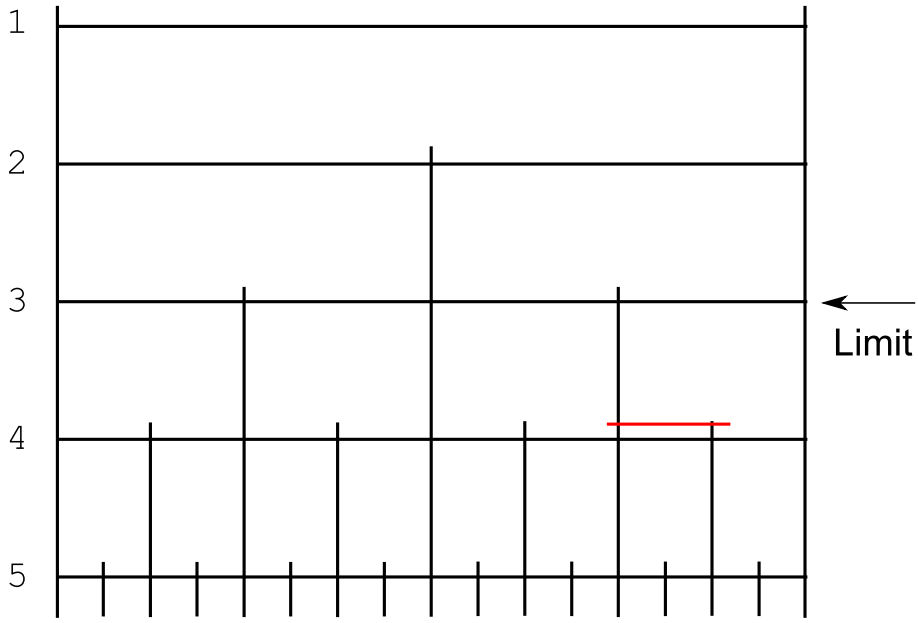


Figure 4.3: Illustration of max depth calculation

Constructing a quad tree from a border is a simple process of filtering out points uninteresting to the newly created node from the parent's list and it has to be done only once. Points interesting to a node, are those that are either inside the subspace represented by the node, or are connected to points inside the subspace. Thus we are using a special case of quad tree called edge quad tree, which partitions if there is an edge passing through the subspace represented by node.

We have already described the process of finding good placement points for fleets. The method we supply for buildings is slightly different. It is a good policy in Defcon to have buildings reasonably spread throughout player's own territory, because attack can come from any direction, especially a nuclear attack. Buildings also require a minimum distance between each other before they can be placed, so this approach is mildly suggested to player. Putting buildings as close as possible to the enemy territory (see figure 4.4) is still not a sound tactical decision.

Quad trees over placeable areas provide us with a simple access to potential placement locations. We iterate over them to find locations significantly distant from other buildings either already placed, or those for which we are preparing a list of locations at the given instance. Sometimes, it is not possible to achieve an ideal spread of buildings so we relax the suggested distance minimum by an epsilon amount. The resulting list of locations leads to very much uniform spread of buildings. The one seen on figure 4.5 is an actual result of this approach.

4.3 Summary

In this chapter, we have shown our methods of map analysis. First showing how we recognize various areas of various type, then explaining options for the underlying methods of access, and finally discussing our use of quad trees as means for approximation of reasonably good fleet and building placements.

Figure 4.4: Illustration of a bad building placement not protecting the cities well

Figure 4.5: Illustration of a good building placement (actual result of the proposed approach) protecting the cities much better.

5. Multi-agent systems and PogamutDefcon

5.1 Introduction

Real-time strategy games are perfect environments for multi-agent systems (MAS). MAS provide a very useful perspective for Defcon. We can see the main AI, all fleets and all buildings as individual cooperative agents fighting against the agents of the other team. Through this perspective, agents can be more self-reliant, but they also require means of communication between each other to coordinate their moves, especially because they usually have limited information about the world.

In this chapter, we are going to discuss MAS in general, and their application to Defcon as well.

5.2 Typical Multi-Agent System

MAS contrast with monolithic AI systems mainly because of their decentralized approach to goal achieving. A typical MAS would have the environment running as one process, and the agents as another. Agents would be fed with their perceptions and they would respond with their chosen actions. These perceptions may not reflect reality with absolute certainty and the actions can fail too and agents have to compensate. Cooperating agents are also likely to share the information in some agreed upon manner, so they can somewhat mitigate possibly faulty sensor readings, if they happen to take readings of a same area and also broaden information for action selection.

Cooperation is often achieved through elections (voting) or auctioning. Both are heavily researched subjects in the area of MAS as exemplified by structure of talks at MAS conferences like Autonomous Agents and Multi-Agent Systems (as of 2011). Before the elections or auctions begin, agents make propositions to other agents, which they then evaluate and assign their preferences to.

Elections are a process during which a group of agents try to come to a best decision of those proposed. Many voting protocols have been proposed with algorithms of different complexity and with different level of satisfaction of agents' preferences. Indeed, the choice of voting protocol affects the results of voting. Classic examples of voting protocols are simple majority, Borda, approval etc.

Auctions differ from elections in that the agents offer some service or set of services and others bid on the service or services. Some basic auctioning protocols are English, Dutch, Japanese, sealed first-price, Vickrey's or combinatorial. They do differ in algorithmic complexity, optimality, level of risk, and whether agents know other agents' propositions etc. Auctions are rarely used in real-time strategy-like environments, because agents in team typically have same goals and their success is therefore measured the same way as success of the whole team.

We are not going to discuss elections and auctions more deeply in this thesis, because they are not the main subject. We are just providing an environment for

utilization and research of these group decision methods. We believe that Defcon could be a very interesting platform for elections. Fleets can suggest to each other what they think is the best course of action from their point of view and then vote to select the best one of them.

5.3 Multi-Agent System for Defcon

Although agents' reasoning is isolated, the MAS architecture we decided to use in Defcon slightly differs from the ideal MAS. It is true that the means of achieving these goals lie upon agents themselves, which is maybe the main feature of proper MAS, but uncharacteristically for most MAS our implementation uses shared representation of the environment: the worldview. This can be rectified in the future of course, but the incoming perception cannot be easily tied to sensors of any single agent or unit. We could choose to filter information for each of the agents using the outside knowledge of the actual range of their own unit or building type, and we agree that it is a possible subject for future work. With this feature we could simulate the results of losing a direct command etc.

Because the main agent is responsible for creation of fleet and building agent, it contains a reference to fleets manager and buildings manager, which keep fleets and buildings accessible more easily and also trigger updates of controlled agents.

Also the main agent takes care of the mailbox for communication between other agents, but it is not meant to modify any of the messages, although it is still possible.

5.4 Pros of MAS Paradigm

In this and the next section we are going to discuss some advantages and disadvantages of MAS paradigm. First let's start with the pros and in the next section, we are going to discuss the cons.

5.4.1 Modularity

The MAS paradigm allows for a more modular design of AI, which brings all the advantages it brings to general programming practice. As a result, the internal behaviour of an agent is much easier to design and debug.

5.4.2 Logic Background

Strong logic background is maybe one of the best features of MAS paradigm. It allows the AI designer to show whether some intended features are actually present in MAS. This field is so important to MAS, that it is as prominently featured at relevant conferences as elections and auctions.

Various logics can be employed to describe system's evolution in time. Agent itself can use various logics for inferences of its deontology, commitments or his own future states.

5.4.3 Game Theory

In MAS game theory is more accented, than in most other AI fields, because it helps understanding cooperation and competition in decisions of relevant agents. Such understanding then helps agents improve their decisions, as the programmer can actually verify, and possibly find even better options, that may be more optimal and stable actions.

5.4.4 Agent Oriented Programming

Agent-oriented programming (AOP) paradigm can be considered an extension of the Object-oriented programming (OOP) paradigm. The role of objects is replaced by agents, who are usually objects as well, but are even more autonomous and individualistic in their behaviour. Agents usually have some goal they want to achieve and they themselves engage proactively in the process. Agents have their own state represented as a list of beliefs, commitments, possible actions and such, thus having a stricter semantic of its own parts. This leads to more clearly defined guidelines during decision making process. Nevertheless, it is only useful in appropriate cases, which is true for every other programming paradigm after all.

In order to simplify AOP implementations, an array of AOP languages have emerged. Such languages are GOAL [5], AgentSpeak(L) [26], StorySpeak [14] etc. Some of these languages make a concentrated effort to also provide some logic formalization as described above. We are going to discuss Jason (a framework using AgentSpeak(L) for agent's code) in chapter 7 along with the successful implementation of a set of agents for Defcon using Jason.

5.4.5 Robotics

Groups of robots are ideal real life physical examples of applied MAS. Robotics have inherently faulty sensors, coupled with limited information about the environment, fallible actions etc. Virtual, purely software, MAS can be used as a testing environment for robots prior to the application in the real environment, reducing costs, while allowing to test many of the required features.

5.5 Cons of MAS paradigm

The MAS paradigm is not without its more difficult aspects. We are going to discuss these in this section.

5.5.1 Debugging Cooperative Eefforts

Decentralization moves the complexity to communication and negotiation with other agents. Making sure that agent behaves accordingly after every possible exchange of information with other agents may prove to be difficult. Sometimes the agent may run into extremely rare situations that are not apparent to be possible just by reading through the code.

5.5.2 Higher Resource Consumption

A single, perfect monolithic AI always performs at least as good as any group of cooperating agents. Monolithic AI uses less resources, both memory and computational, than group of cooperating agents. Nevertheless, if we have multiple processors at our disposal, then offloading of the execution to multiple processes may lead to a better performance.

A special case of resource demands are AOP programming languages as they are almost exclusively interpreted. String-based interpreted programming languages are processed by interpreters at real-time, without requiring any previous compilation. It has its massive advantages, like being able to change code at real-time etc., but string-based interpreting also leads to a much higher amount of string operations, which tend to slow down applications a lot, if there is too many of them. Programs executed from compiled native code, or at least a middle-end interpreted code like Java byte code, are always going to be more efficient.

5.6 Summary

In this chapter, we have discussed the multi-agent system paradigm. We have explained why it is so useful for Defcon AI and why is it useful in general. We have concluded the chapter by listing some of its pros and cons.

6. Proof-of-concept AI For Defcon Using PogamutDefCon

6.1 Introduction

In order to examine usefulness of PogamutDefCon, we decided to implement two proof-of-concept examples. We are going to discuss the first one of them in this chapter. A MAS inspired set of agents written purely in Java. If you have not read chapter 4 yet, then we urge you to do so, as we are going to build upon many points we have raised there.

6.2 Main Agent

Implementation of the main agent is rather direct and simple. Pre-processing consists of three steps:

1. *GameMapInfoPolygons* is used to collect areas.
2. Areas are passed to *QuadTreeManager*, which constructs and holds quad trees of over areas.
3. Finally spawn points with intended targets are constructed in *ClosestPointsLookUp*.

Once the pre-processing is done, the agent switches to the ship placement mode. The ship placement mode performed by fleets' manager uses spawn points calculated by *ClosestPointsLookUp* as described in 4. The process is not as intuitive as it might seem at first. Commands sent to Defcon are not carried out immediately and have to pass through synchronization in *PogamutJBotSupport*, and therefore fleets' manager cannot assume, that it has completed successfully until it has actually received an event from worldview that confirms it. This may be necessary if the programmer avoids using *ClosestPointsLookUp*, as we don't want to force her to use it to get non-conflicting spawn points. Fleets' manager then reports back through a listener providing a list of successfully placed fleets, whether the whole amount of fleets requested was placed to any of the given placement points and even returns the given initialization object. This initialization object is then passed to fleets' manager with the request for placement. In case of this example implementation, we pass the target location, for which *ClosestPointsLookUp* computed spawn points, as an initialization object for *MixedFleetAI*, which is the single fleet agent AI we have implemented. *MixedFleetAI* does not choose precisely this direction, but tries to improve upon it. We will discuss it later on this section.

Building placement starts after fleet placement has been finished. Building placement is simpler, and it represents slightly different approach to placing. User just creates a list of placement points or uses *BuildingPlacementProvider* class. We used the *BuildingPlacementProvider* class to generate them. Its method of finding possible placement points also uses quad trees, as we have already

explained in the last section of chapter 4. After placing all of the radars, nuclear silos and airbases (Defcon returns 0 as the amount of placeable buildings for all types), the main agent retrieves a list of all visible buildings from worldview and assigns new instances of proper AIs to each of main AIs placed buildings. Proper AIs are selected based on the type of a building. Radars receive none, nuclear silos receive *SiloAI*, and airbases receive *AirbaseAI*. We chose this different approach, because we assume the building agents do not require any initialization. Also we might pick a single preferred method later on after a longer term and more extensive use by variety of programmers.

All three types of agent AIs implement interface *IUnitAI*, which is inherited by the *AbstractAI* class, which implements some methods useful for all AIs. *MixedFleetAI* implements *AbstractAI* directly, while buildings inherit *BuildingAbstractAI*, which again, adds some of functionality useful for all building AIs. All of the interfaces and abstract classes mentioned in this paragraph are part of PogamutDefCon.

6.3 Silo Agent

Silo has two functions: air defence and long range nuclear strike. It spends most time in the air defence mode, protecting against enemy aircrafts (fighters and bombers) and also against nuclear missiles of all types: short range from submarines, medium range from bombers and long range from nuclear silos. During the game of Defcon, no player should change silo's state more than twice, because it would be a waste of time and defensive capabilities it would provide by protecting against air targets. Therefore agents have to make only a single decision: When should I change to nuclear strike state? In our implementation we chose a fixed time, which is about a minute after Defcon 1 is put into effect, while running on the fastest game speed. It has proven very effective against internal AIs, because the first volley tends to catch the enemy AI of guard while it itself is switching to nuclear strike state. Some optimization could be done with regards to defence against short range nuclear missiles from enemy submarines though. The enemy AI seems to strike with its short range missiles from submarines at the precise moment our silo's agent changes the mode to nuclear strike. The consequences tend to be quite deadly, but our fleets tend to very successful as well.

6.4 Airbase Agent

Airbase agent is slightly more complicated than the silo agent. Airbase has three kinds of resources: i) fighters ii) bombers iii) medium range nuclear missiles. Fighters can attack bombers, battleships, carriers, nukes and other fighters. Bombers can attack surfaced submarines, battleships and carriers with their basic attack, and after Defcon 1 they can be loaded with medium range nuclear missiles and attack cities, and buildings as well.

Up until Defcon 1 the agent launches only fighters at targets that are in range. When Defcon 1 is hit, the airbase launches all bombers at random enemy cities for a nuclear strike. Bomber launching repeats if there are still nuclear missiles left

and bombers present at the airbase. This situation is possible because bombers return after their mission back to airbase or carrier, where they can load a new missile.

6.5 Mixed Fleet Agent

We use only one type of fleet although players can create a fleet of an arbitrary composition as long as they still have enough units at their disposal. The fleet composition of a mixed fleet agent is balanced, containing two of each kind of naval units. That is two submarines, two carriers and two battleships.

After the initialization, as described in the main agent description section 6.2 of this chapter, the fleet receives a spot in the intended target naval placement area of the enemy. The agent does not take this target strictly, but tries and optimizes it slightly using the following method.

A centre of gravity is calculated for the cities of the owner of the closest city to the passed target location. It is a very general approximation, which we then use to refine the fleet's first movement target. The refinement process consists of finding the closest location accessible to naval units, while testing points from the cities' centre of gravity distant by an epsilon vector with length of 1. Once the closest location is found, the process then checks a few couple samples in direction of epsilon vector, whether they are still accessible. If not, then the closest location is reset to the next accessible one and the process repeats. The count of added samples depends on the parameter passed to the tracing method, which represents the maximum distance samples have to go, while being accessible, before the resulting location is accepted (see figure 6.1). The next step is to find a possibly better candidate.

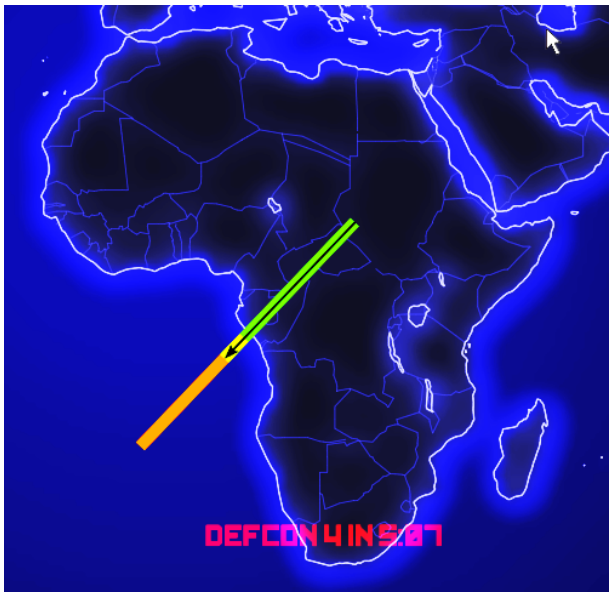


Figure 6.1: Illustration of tracing used during refinement of the fleet target location

This candidate is calculated based on the current location (spawn location) of the fleet and the aforementioned centre of gravity. What we get is a direct path

to the enemy's cities centre of gravity. This solution might be better because enemy's fleet placement areas are not necessarily enveloping enemy's coastline in its entirety and we might then lose a viable option.

Now that we have two optimized movement target locations, we can choose between them. We always choose the closer one among these two and we set it as a movement target once Defcon 4 has been announced. Next step depends on whether fleet reaches the movement target before or after Defcon 1. If it reaches the movement target before Defcon 1, the fleet just rests at the target position until Defcon 1. Once it has been set, the arrival time is noted. If the fleet reaches the target position after Defcon 1, the arrival time is recorded immediately.

From that moment, the fleet begins its nuclear strike, sending bombers with nuclear missiles and striking with missiles from submarines at the same time. When all missiles have been launched or the fleet has been at the launch location for more than more than a fixed amount of game time, the fleet changes states of all units to an antisubmarine mode. The fleet then tries to find the closest enemy fleet, with either a carrier or submarines, as they carry nuclear missiles, and pursues it. If it cannot find any, it chooses a random fleet. If it cannot find any either, it chooses its original location (spawn points), which is optimized in the same manner as the target location the fleet has used for attack.

Finally fleet rechecks its target periodically to see whether the target fleet is still near its original location. If not, then it tries to find a new target fleet or chooses its optimized original location as described above, but with AIs own cities gravity center.

6.6 Note on Resource Requirements

The execution itself has a very small impact on computers resources. When compared to the Jason implementation, which we are going to discuss in chapter 8, it is almost negligible. The game playing performance of the purely Java-based set of agents against the internal AI of Defcon is the main topic of the chapter 9.

6.7 Summary

In this chapter we have described an implementation of our Java-based AI for Defcon. We have discussed how we build upon the results of map analysis tools that we have studied in chapter 4 in order to optimize the results even further inside our main agent's code. Then we have analysed the other three important types of agents we use and their implementations, starting with the agent for nuclear silo, continuing with the agent for airbase and concluding with the mixed fleet agent. Finally we have finished with brief notes on the performance of our Java-based AI.

7. Jason

7.1 Introduction

Before describing the implementation of Jason solution, we have to expose and study Jason [7] itself as there are many concepts to understand, in order to use it properly. We will start with the Belief-Desire-Intention architecture, then move to AgentSpeak(L), which is the language Jason builds upon and finally we will get to Jason, its implementation, features etc.

7.2 Belief-Desire-Intention

Belief-Desire-Intention (or BDI) model of human reasoning has been proposed by Michael Bratman in [8]. Later on he participated on the BDI software architecture. The basic idea of BDI is simple a formalization of agent's reasoning based on four elements:

- agent's *beliefs* i.e. the current state of the environment, or more precisely the state agent believes the environment is in,
- *desires* i.e. what would the agent like to achieve, with non-conflicting desires being goals to be adopted,
- *intentions* i.e. adopted goals in process of being achieved, and
- library of *plans*, that the agent can follow.

To explain the second point, let's use an example. Imagine an agent desiring to go to a cinema and also desiring to go to a theatre, with both the play and the movie running at the same time. These two desires are incompatible, because they conflict with each other. The agent cannot be both at the cinema and the theatre. The agent has to choose only one, making it a goal and choosing a plan to follow to achieve it, making it its intention.

BDI architecture also proposes steps and their precise order during agent's deliberation. The algorithm begins with the initialization consists of assigning initial beliefs and intentions from selected goals. After the initialization, it keeps repeating the following set of instructions up until all intentions have been satisfied. Following pseudocode, taken and adapted from [33], describes the reasoning cycle:

1. Obtain perception.
2. Acquire or drop beliefs.
3. Find a new list of achievable intentions, which have not been achieved yet, from current intentions considering current beliefs.
4. Pick the best intention.
5. Find a best plan for it.

6. Perform the next action of the plan.

Subsequently numerous logics have appeared, with the goal to model the evolution of BDI agent's state. Examples of these logics are BDICTL [27], which is a form of multi-agent computational tree logic (CTL), or Logic of Rational Agents (LORA) [33].

One of the major BDI languages is AgentSpeak(L) by Anand S. Rao, which we are going to discuss in the next section. Others are e.g. GOAL [5], 3APL (An Abstract Agent Programming Language) [17] or 2APL (A Practical Agent Programming Language) [10].

7.3 AgentSpeak(L)

"AgentSpeak(L) is a programming language based on a restricted first-order language with events and actions" [26]. It is agent-oriented, declarative, with a Prolog-like syntax and concentrates on the specification of plans and initial beliefs and goals. Each plan is written in the following form: $event$ **:** $context$ $\leftarrow$ $plan\_body$. For an example of such a plan see figure 7.1. Triggering events come from perception, initial goals and intentions, which either add or remove a belief or succeed or fail a goal. Context is a boolean expression, which has to be true, in order for a plan to be valid in the current state. The body of a plan is a sequence of terms, where term can add a new goal, add or remove a belief or call an action. Although BDI architecture is apparent from the syntax, it is the reasoning cycle, which actually categorizes it as such as you can see in figure 7.2.

$$
\begin{aligned}
+!start2 \ &\mathbf{:} \\
&\quad running \ \& \ nearest(corral\_ally, X, Y) \ \& \\
&\quad close\_enough(X, Y, 17) \ \& \ team(Team) \ \& \\
&\quad \mathbf{not} \ allies\_nearby(Team) \\
&\qquad \leftarrow \ .relevant\_plans(\{+!start1\}, \_, L); \\
&\qquad\quad .remove\_plan(L); \\
&\qquad\quad !!start2.
\end{aligned}
$$

Figure 7.1: Example of a Jason plan (not relevant to Defcon)

AgentSpeak(L) often serves as a basis for other languages and BDI implementations as it leaves some of its aspects undefined. Among these are mainly selection functions. These incldue the event selection, plan selection and intention selection.

7.4 Jason

Jason is not only a language, building upon AgentSpeak(L), but is also a framework for Java, which implements the back-end of the BDI agents. It is one of the most prominent implementations of AgentSpeak(L), with a wide user base, with a wide range of extensions and optional integration to several middleware platforms such as Moise+ [18] or JADE [3].

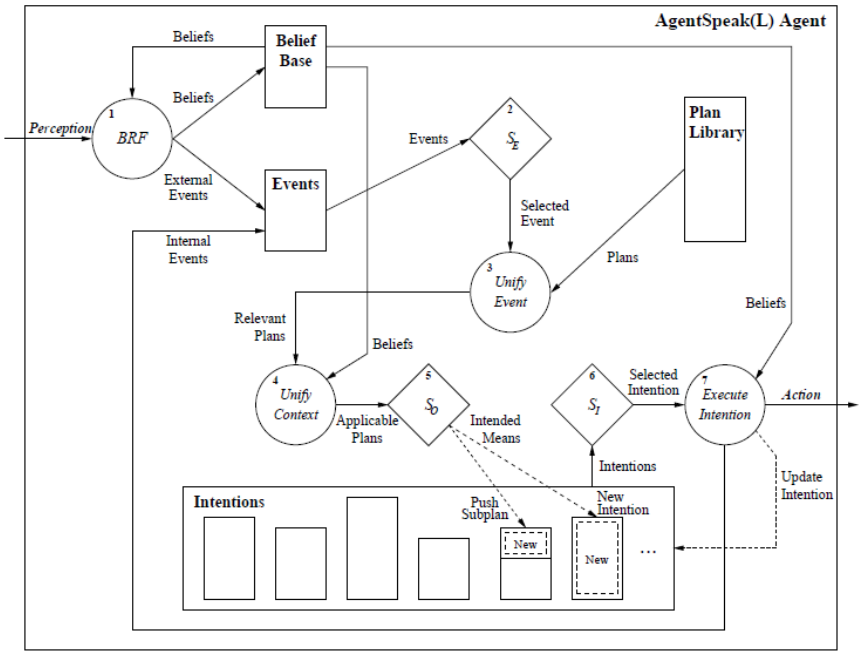


Figure 7.2: Jason reasoning cycle *Source: [7]*

Jason has a list of useful features, which we are going to discuss now. It extends AgentSpeak(L) with a Prolog implementation. It allows for a high degree of customization, allows for custom belief bases, actions, selection functions etc. It also provides debugging tools, which can be used to observe inner states of agents, showing everything important from the current state of belief base, inferential rules of Prolog, actions currently being performed, plan options, and finally a queue of events. Let's examine some of the important constructs of Jason offered to a programmer.

7.5 Agent

Agent is the main class for all agents, because it hosts the interpreter (transition system), belief base, internal actions (explained later on in this section) etc. All three selection functions (the event selection, plan selection and intention selection function) are to be implemented here. For example, we may want to always prefer an event warning our agent about incoming danger. We may also want to prioritize intentions, like the one, which has been created by the aforementioned warning event.

Modifications of these selection functions are quite often, as source codes of many Jason agents would grow rather complicated. On the other hand, modifying these functions leads to a more obscure design, moving more of the decision making process from the agent, where we intend it to be from the start, back to Java code.

7.6 Agent Architecture

In Jason, all main actions called from the agent's Jason source code are passed here (through calling the *act* method), semantically parsed, and performed in the environment. Perceptions pass through agent as well (through the *perceive* method) and finish the encapsulation of the agent as described. Agent architecture is the single class that the Jason agent has to actually implement in order to work. The belief base, transition system and agent can use a default implementation.

7.7 Internal Actions

Environment actions do not allow unification of variables, because they have to have all of its terms *grounded* i.e. actions cannot have any variables in it. They are not meant to communicate back anything apart from success of failure. Implementing all required functionality inside Jason source code is unrealistic and inappropriate in most cases. Let's consider the classic A* algorithm as an example. Prolog implementation hardly offers an optimal solution with regards to speed of computation. Prolog is designed for knowledge inference through unification and A* has some aspects, like the heap data structure, which are not easily implemented in Prolog.

An internal action is a combination of a term and action. It can set restrictions on arguments, but the actual restrictions are entirely dependent on the internal action's implementation. For example, an action may support only three arguments (has arity of three), of which the first one is a term with a specific functor and two grounded arguments of type *number*, same restriction on the second argument and third argument a non-unified variable. This is exactly what a basic internal action for A* algorithm could look like. After calling this internal action, Jason then instantiates a class implementing the *InternalAction* interface having the same fully-qualified name as the action. The A* algorithm itself can be implemented inside this class and unify the path back to the third argument of the action.

Internal actions are one of the most basic constructs of Jason and most Jason applications have at least a few of them.

7.8 Belief Base

Belief base is an integral part of BDI architecture. To put it simply, it contains a set of literals representing the whole of agent's beliefs and takes care of agent's perception. When a new perception is added to the belief base, the belief base raises an event, which might possibly add a new intention and instantiate a new plan if there is such a plan inside the plan library, which has it as its triggering event and has a valid context.

The basic implementation uses a standard Java *HashSet*, which is sufficient in most cases. There is a wide variety of possible implementations of belief bases though. Programmer may want to use an actual database as a belief base, possibly creating a form of shared or a persistent belief base. As an example of such

approach one might consider *JDBCPersistentBB* class, which uses JDBC (Java DataBase Connectivity) API for connection to a relational database.

Simple modifications are possible and useful as well. We might, for example, want to make sure there is always at most one instance of a belief of a certain type. This type might be entirely arbitrary, depending only on programmer's definition. Let's say we want only one belief that represents agent's current location to be present in the belief base at any given time. This is actually rather easy, because we can use a unifier. Unifier tries to perform unification (most general, or MGU) between two terms. If we have agent's location as a compound term of arity of two, then we have to make sure, that there is always only one belief that has a functor used to represent the agent's location. The two arguments can be anything, unless we want to make sure, that it is also a valid location. In Prolog we would use non-unified variables or the so-called anonymous variables (without a name) for each of the arguments and in this case the solution is same. This masking term, will unify with all agent's location terms and none other.

7.9 Environment

Jason provides a class that can be inherited and then used by Jason as an environment for agents or a wrapper for it. One of the provided implementations is *TimeSteppedImplementation*, which can be used to wrap or create an environment programmer would like to use. It is not necessary though and may be cumbersome and redundant, if the environment itself is being access by agent through some more complex means.

7.10 Execution Control

Agent's execution may sometimes have to be controlled in order to work properly in the given MAS. For example, we may have a simultaneous turn-based environment, in which all agents must first choose and commit an action, before the environment or just a set of agents may move forward in their life. Some agents do not require synchronization and thus can run independently as is the case of real-time strategies. It is always important to choose an appropriate one.

7.11 Runtime Services

Runtime services can be used to manage agents, while the system is running. In our case, we have to work with runtime services heavily, because agents are created and destroyed dynamically. Defcon agents are created by the main agent after buildings or fleets have been placed, and are destroyed when their buildings or fleets are physically destroyed in game.

7.12 Running a Jason Multi-Agent System

In order to simplify launching of agents from Jason MAS setup files (.mas2j files), authors have provided classes that automate the process. Programmer

does not have to concern herself with them, if she chooses to go with the standard architectures as those are already in place.

Jason MAS setup files have a nice advantage of simple initialization of agent's objects like belief base and such. In .mas2j files user can set what kind of architecture she would like to use. The basic MAS architecture options are centralised architecture, SACI (Simple Agent Communication Infrastructure) architecture, JADE (Java Agent DEvelopment Framework) [3] architecture. SACI and JADE architecture are decentralised architectures. These architectures wrap user's agent architecture. Next, the programmer may or may not specify what environment and what execution control she would like to use.

Finally a list of agents with their names, source file names and parameters is supplied. Parameters consist of the actual agent architecture, which is wrapped by one one the MAS architecture's agent architecture to allow for decentralization of the whole MAS etc., agent's intended belief base and agent's class.

There are two typical ways and one rather atypical way of running a Jason project. One is running it through Jason IDE, second through Eclipse plugin. The first option works well, but Jason IDE is not the best for Java development as there are better, far more advanced IDEs like NetBeans [22] or Eclipse [13]. The second option is quite comfortable, but it has its issues as well, since the Jason plugin for Eclipse is not entirely bug-free. The syntax highlighting of Jason code is one of the main examples, as from time to time the colors disappear.

The other option to run Jason application is to use the provided classes for creation of an Ant buildfile, which handles the launch and after launch cleanup as well. Finally a programmer can completely circumvent all of these and use one of the runners provided for most MAS architectures.

7.13 Debugging

Such framework as Jason cannot work well without some means of debugging. There are two standard ways programmer using Jason can do it.

First is the standard tracing into code at the specified point using a debugger provided by the IDE and looking for incorrect assignments, incorrect method calls etc. This method is not perfect for real-time systems, like MAS, because different processes (or threads) have to interact fairly often, thus leading to race conditions. Race conditions are a result of at least two processes performing task at the same time and interacting at some point, but not taking into account all possible states or events of the other process, possibly arriving at invalid results. Programmer therefore has to use her knowledge of the multi-threaded programming in general to see, whether such states are even possible and would make a difference.

Second is logging, which is also a standard method, and even more standard for MAS at large. Logging minimizes race conditions as it only outputs some information deemed by the programmer to be a sufficient evidence for her to deduce where the bugged code is so there is no need to stop the evaluation. In an unfortunate case that logging does eliminate the race conditions, then the programmer faces a much more difficult issue to solve, since she has to use her own skill to find the problematic code just by analyzing the whole code step by step.

The added feature for debugging of Jason is the inspection view (see figure 7.3), which allows the programmer to analyze the mental states of the agents present in the Jason controlled part of the MAS. The programmer can stop the execution of the agents at any given point and examine their beliefs, currently queued events, actions, for which the agent waits to be performed, intentions, and Prolog rules for inference.

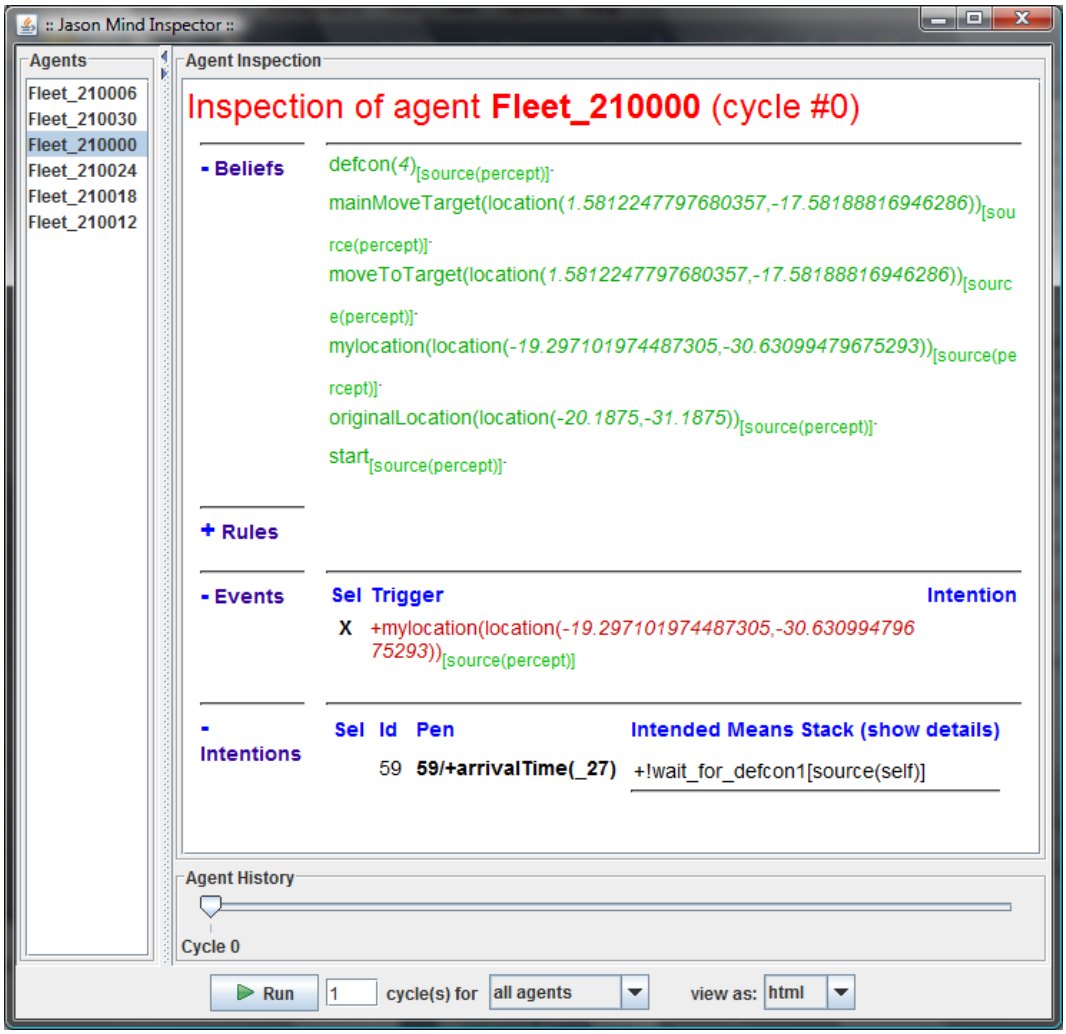


Figure 7.3: Jason's inspection view

7.14 Summary

In this chapter, we have described Jason as a Belief-Desire-Interest framework for multi-agent systems. We have discussed the Belief-Desire-Interest software architecture, its origins in a theory of human reasoning and reasoning cycle. Then we have talked about AgentSpeak(L), which is a classic BDI language and shown that it is a language, upon which Jason builds. Next, we have discussed the important classes of Jason, which we are going to reference later on in this thesis. We have introduced concepts the programmer is supposed to use in order to launch Jason agents and MAS. Finally we have discussed debugging of Jason, which is also useful subject.

8. Jason based AI for Defcon

8.1 Introduction

Jason is very popular in academia. We have decided to combine its power with PogamutDefcon, because we wanted to extend options for implementation through MAS we have already provided. To test our support, we have implemented our second AI for Defcon using Jason.

In the first part of this chapter we are going to explain our use of Jason, provided classes for interfacing etc. In the second part of this chapter we are going to analyze the AI itself, from the infrastructure code in Java to the logic code in Jason.

8.2 Connecting Jason with PogamutDefCon

In order to simplify launching, yet avoiding the typical methods of Jason, we have combined the real-time instantiation of Jason objects with one of the basic runners. We chose the *RunCentralisedMAS* runner as our template, because we wanted a centralised architecture for Jason agents. *RunCentralisedMAS* runner takes the .mas2j MAS configuration file and instantiates all agents it finds inside. We wanted to make the code as short as possible, so we have used Java reflection framework to access some of the private fields. Otherwise we would have to copy and paste the runner's code. It is slightly dirty from the programmer's point of view, yet it makes the code a slightly more readable. Still, we had to modify *RunCentralisedMAS*'s functionality slightly. Its handling of priority of agent's thread setting and loading of general settings, like logging, from files inside a jar file, which is a file compressed with zip compression combined with some useful information for Java to handle it as a library or a complete application.

For example, if the original *RunCentralisedMAS* loads in a jar file mode, it uses only a default logging scheme through console (Jason's console to be precise). This is very limited for our purposes, since we would like to allow user to set Jason's output to an arbitrary destination. On the other hand, if *RunCentralisedMAS* runs in a simple the non-jar mode, only with compiled classes, it tries to look up a file named *logging.properties* in the working directory in which Java executes the application. We modified the process of setting up the logger to check for this file inside the jar file as well. If there is a *logging.properties* in the root directory of the project during the jar building process, then it is included inside the output jar file for Defcon. Our implementation of the runner, called *RunDefConCentralisedMAS*, checks for the *logging.properties* in the jar file, and if found, configures the logging according its content. Some possible output targets are text files and, of course, the aforementioned console.

The second important feature missing from the standard *RunCentralisedMAS* is an option to set priority of a thread agent runs in. This is crucial, since it is one of the important variables for a programmer, who wants to manage resource demands of the agent.

Now, that we have our version of the *RunCentralisedMAS* customized for the needs of our PogamutDefCon, we can move on and use it to create agents. The

.mas2j file, describing the intended Jason MAS, and packed inside the jar, is slightly different from the usual one, because we do not expect any other agent than the main agent to be included in it. Other agents would otherwise be instantiated immediately, which does not reflect the approach expected from the user as described in the next section 8.3 dealing with the implementation of the main agent.

We have separated our implementation of the Jason bridge from PogamutDefCon project. We call the new project PogamutDefConJasonInterface. This library contains a following list of important functionality:

- *RunDefConCentralisedMAS* class
- Some example internal action classes
- *GeneralAgArch* abstract agent architecture class implementing basic useful functionality
- Basic interfaces that agent architectures can rely upon, when they want to interact within the Java part of the agent

Now that we have explained our bridge between Jason and PogamutDefCon, we can move to the main agent's implementation.

8.3 Jason Main Agent

First, let's describe the way this implementation integrates the Jason interface into its code. Once the main agent code enters the *preGameLogic* method of *JasonBotLogicController*, the *RunDefConCentralisedMAS* is instantiated and initialized. It takes the provided .mas2j and instantiates the main Jason agent for the main agent.

The initialization is performed in a different thread, because Jason takes the thread for its own execution, after the initialization is done. The main agent code then waits until Jason creates an agent named "main" as this is the agent from the .mas2j file.

The main agent is very similar to the main agent of chapter 6. Short recapitulation of its execution follows:

1. Calculate borders of areas (or load them from cache)
2. Construct quad trees
3. Find appropriate fleet spawn points, place fleets and assign each of them an instance of proper agent
4. Perform next three steps in random order
5. Find appropriate silo placements and place silos
6. Find appropriate airbase placements and place airbases
7. Find appropriate radar placements and place radars

8. Assign silos and airbases each of them an instance of proper agent

9. Stop

Since we wanted to show the possible user an example of the main agent written using Jason, we moved the definition of this process into Jason source code. Jason source code is located within *main.asl* file. The code is very high level, practically only mirroring the above mentioned process.

Still, in order to showcase some aspects of the Jason code, we have randomized the order of building placements using only it. Not only that, we have used an advanced concept of Jason by using a unified variable as an action. Once we have the randomized list ready we use a plan, which keeps trying each of the actions until all of them return true (they are true as a term), reporting that they successfully placed all possible buildings.

Assignment of the Jason agent takes place at the same time as the assignment of the agents from chapter 6. The difference is that the main agent asks *RunDefConCentralisedMAS* to create a respective Jason agent, which is then passed to the stripped down version of the agent from chapter 7 in form of an agent architecture crafted specifically for this fleet or building. The stripped down version also takes care for the respective Jason agent's disposal once its fleet or building has been destroyed, when it is reported from either fleets' manager or buildings' manager. For the duration of this chapter, let's call these stripped down versions simply *Java agents* for the sake of simplicity.

Sadly, we had to leave the Jason agents for buildings out, because Jason could not handle them fast enough. One main agent, about six fleets, four airbases, six silos make up seventeen agents in total. That number is way too high for a reasonable execution as we have verified by experience. So, altogether we have seven Jason agents, which is still only barely manageable. We have not found a way to optimize it well enough and it is definitely a valid point for future work.

Here it is also important to discuss the belief base of the main Jason agent. We use a modified belief base that allows user to specify a list of terms that have to be unique in the belief base with regards to unification. For example, one of these terms is *defcon(_)*, meaning that there is only one term with *defcon* as a functor. If we have put a number 5 as the argument, then there would be only one term *defcon* with argument equal to 5, not enforcing any uniqueness. Therefore, it works like an ordinary Prolog unification. Why did we use this belief base with the unique beliefs list even for the main agent, which does not have any need for its own location? We wanted to simplify creation of Jason fleet agents so we could just clone the main agent's belief base and use it for the mixed fleet agent.

Let's move to the Jason mixed fleet agent implementation now.

8.4 Jason Mixed Fleet Aagent

As is the case with the Jason main agent, the mixed fleet agent uses virtually identical design to the Java-based mixed fleet agent from the chapter 6, so we are not going to discuss the stages of execution, but the interesting points related to Jason.

Let's have a look at the important parts of the Jason code. First we declare a list of Prolog inferential rules to simplify calls of custom internal actions, as their

fully qualified names are unnecessarily long. After starting the agent, the initial *start* achievement goal is added and a first plan is triggered. The fleet agent then announces it is running and waits for Defcon 5 to be added to belief base, as this is a perception the Java fleet agent uses to tell the Jason agent that it is ready to accept its commands and that the optimized target location, has been computed. The Jason agent then waits for Defcon 4 to be in place and then moves the fleet to the optimized target location by issuing the movement order and adding a new goal for the next part of execution.

The more interesting parts come with changing to nuclear launch state, as the Jason agent does that without any direct help of the Java agent. The whole process is done entirely through internal actions for checking for a proper unit type and proper nuclear launch state, filtering out all that do not support nuclear states and then setting those that actually do. Then the agent periodically checks whether the fleet either has spare nuclear missiles, or it has spend long enough time at the launch location. The rest is easily understandable in this context.

The Java fleet agent itself takes care of basic perception like the fleet's own location, launch location arrival time and current Defcon. It also provides the implementation of required methods the Jason fleet agent architecture calls in the *act* method, which takes care of the environment action.

8.5 Jason Building Agents

In order to at least show a direction of Jason building agents' implementation, we included agent architectures for them and basic Java agents as well. We have not actually implemented anything inside Jason building agents code, because after we noticed the speed of execution, we saw that it would not be feasible to run all the agents at the same time.

8.6 Note on Implementation

We have not modified any of the selections functions. As we said in chapter 7 it is often the case though. We believe, that such modifications should be kept to minimum, as they blur the meaning of the Jason code, and make it less understandable for other programmers.

8.7 Summary

In this chapter we have explained our small framework for connecting Jason with PogamutDefCon. Then we have discussed our implementation of a Jason-based AI and shown its relationship to the original Java-based AI.

9. Evaluation

9.1 Introduction

As a final step of this thesis we performed an evaluation of both the AI implemented purely in Java, as described in chapter 6, and the AI implemented using Jason as described in chapter 8. For this purpose, we have created a special testbed, which executes duel based tournaments either between a set of AIs or between the internal Defcon AI and a set of AIs. Finally we will perform a goodness of fit test to see, whether both the Java-based AI and the Jason-based AI are on par with the internal AI.

9.2 Testbed

Our testbed uses Java for implementation and Ant for execution. The process begins by loading a testbed configuration file from command-line parameters. The structure of the configuration file is shown in figure 9.1. Inside the configuration file, user can specify following in random order:

- mode
- number of iterations
- host path
- client path
- host's territories
- client's territories
- players, with each player on a new line

In mode section user can either of the two modes of testing, default one being *Tournament* and the other one *InternalAI*. As already said in the introduction, one specifies a tournament between specified AIs, while the other a tournament against internal Defcon AI. Number of iterations means how many times each battle of the tournament is repeated before it advances to next setup. Host path specifies a location of Defcon, which is to be used to host a server, which

```
Mode:  internalai
Iterations:  2
HostPath:  C:/Games/Defcon
HostTerritories:  0 1 2 3 4 5
ClientTerritories:  0 1 2 3 4 5
Player:  ../01-ExampleBot
```

Figure 9.1: Example tournament definition file

is also true for the case of internal AI mode, even though it does not host any other clients apart from the host itself. Client path is meaningful only in case of tournament mode, and specifies the path for Defcon to be used by the client to join the hosting instance. Host's territories field specifies the list of territories the hosting instance AI is assigned territory from. The specific territory is chosen chronologically for each of the client's territories specified in the client's territories field. All instances, which would lead to a same territory chosen by both players, are skipped. Finally the list of paths to player AIs is specified by a whole set of all player fields.

Once the tournament configuration file has been parsed, the tournament is instantiated and begins to prepare the execution, by preparing the parameters for Defcon instances. After preparation the iteration through players and territories begins. The Ant buildfile *build.xml*, included in the root of the tournament project, is executed for each of matchups. The buildfile uses a property passed as a command-line parameter to locate the directory of the AI, then it calls the *deploy* target from projects own buildfile *build.xml*. Note that this is a prerequisite for a successful tournament. All player AIs have to have their buildfiles with the *deploy* target accessible in this manner and also performing all the necessary steps of compilation, packaging into jar file and deployment into Defcon directory, which is passed as an ant property *defcon.path*. If any of these steps perform unexpectedly, the tournament is not guaranteed to succeed. After a single match finishes, the output data is collected from the standard output and recorded into statistics of each both players, or one in case of matches against internal Defcon AI. Each time a new pair of results or a single result is collected, a callback from tournament is called and through that the results are reported. This callback is meant to be used to record tournament in progress, in case of an unexpected general crash resulting in tournament's termination happens. We record the results in an XML file *result.xml* in the root of tournament project's directory as default.

The information is collected from the standard output of the host's instance, and is put there by *PogamutJBotSupport* after the match has ended. Following the output, *PogamutJBotSupport* then waits one second and closes Defcon.

9.3 Pure Java AI Evaluation

We have evaluated the pure Java AI implementation using the internal Defcon AI. The setup was simple. All territory pairs are examined by setting a complete list of territories for both host and client. Each combination of territories was evaluated twice. We have used default settings for the API with exception of setting the number of internal AIs and territories. Most importantly, this means agent having only a limited information from the environment. The AI player perceives exactly the same things as any human player would perceive being in its place. We used a simple win/loss ratio to evaluate the results.

The pure Java AI results can be found in figure 9.2 and table 9.1.

Total kills	Total losses	**Won**	Lost	***W/L Ratio***
4042096000	3782259886	32	28	53.33%

Table 9.1: Java - Game results

9.4 Jason AI Evaluation

Although we have been more concerned with showing that our work is capable of connecting Jason, we have also performed the test for our Jason AI for the sake of completeness. We have evaluated the Jason AI implementation using the internal Defcon AI. The setup is identical to the pure Java implementation.

The Jason AI results can be found in figure 9.3 and table 9.2.

Total kills	Total losses	**Won**	Lost	***W/L Ratio***
3910970111	3782259886	25	35	41.66%

Table 9.2: Jason - Game results

9.5 Summary and Conclusion

Both of the implemented AIs show that they are able to compete against the internal Defcon AI. The results can be found in table 9.1 and 9.2. The first one even managed to beat the internal AI.

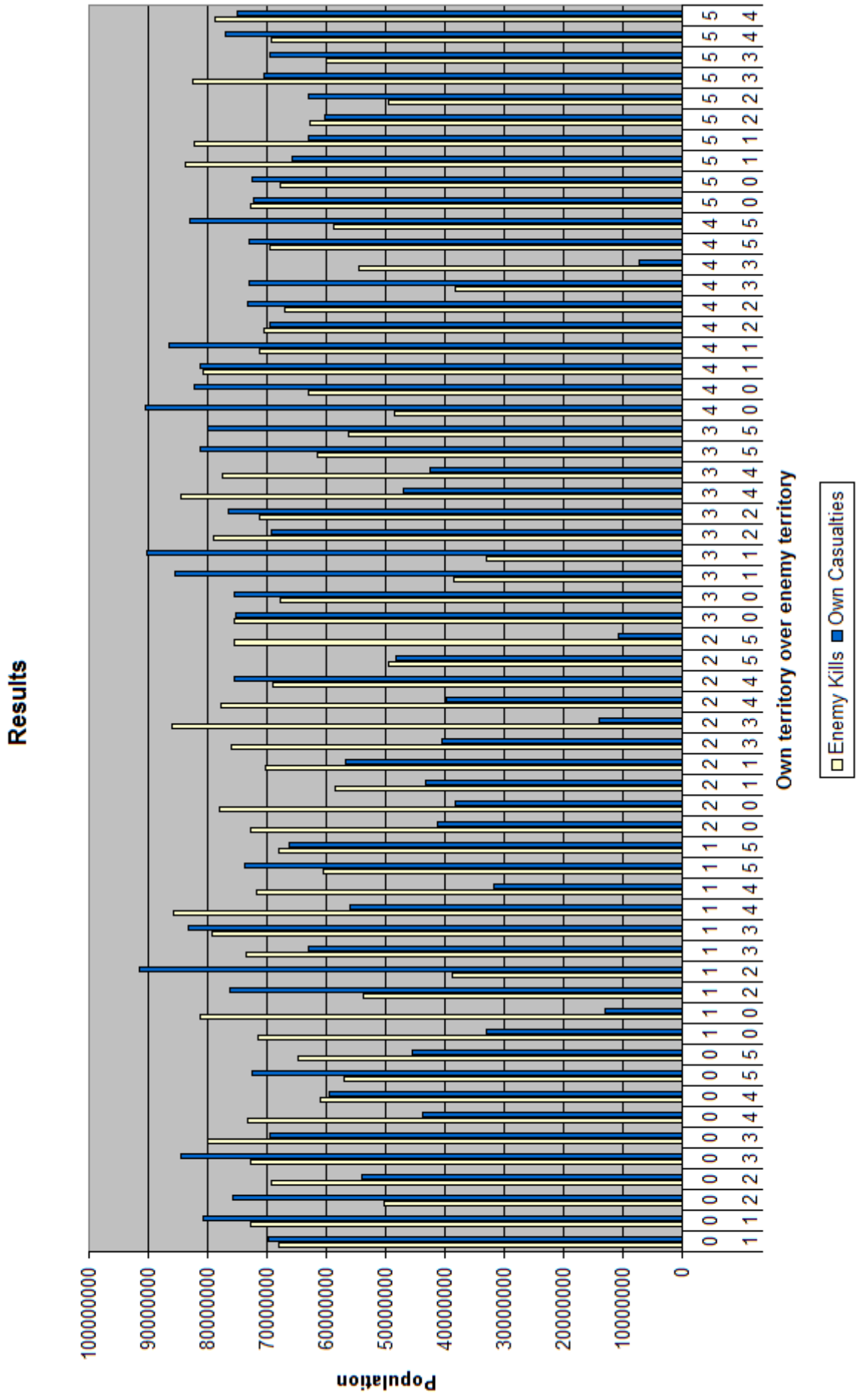


Figure 9.2: Java - Match results

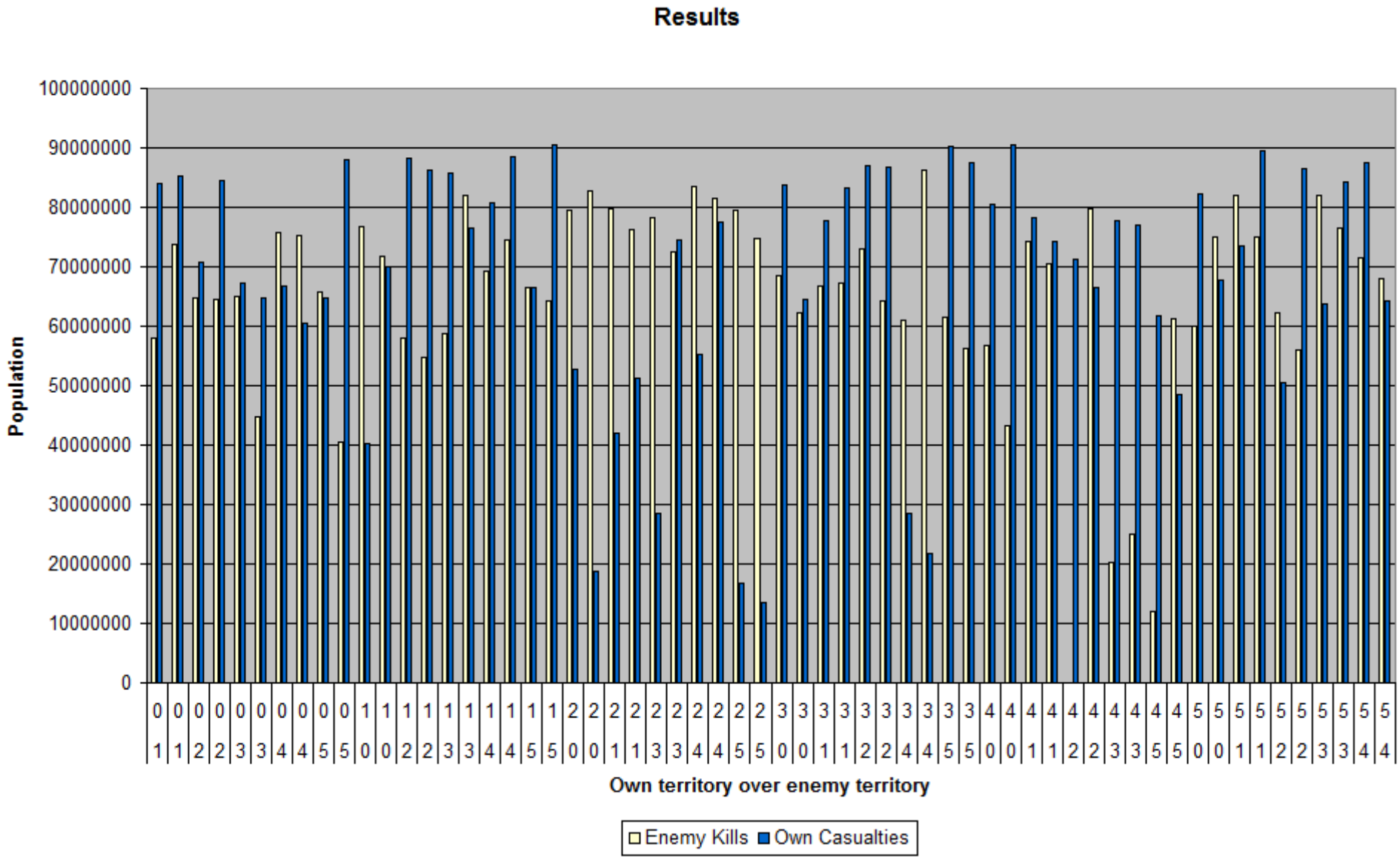


Figure 9.3: Jason - Match results

10. Final Words

10.1 Related Works

CIG 2009 conference hosted a competition in Defcon facilitated by Defcon AI API. Robin Baumgarten himself based his master thesis on it and many papers on various aspects of his AI for Defcon [1]. He had used evolutionary algorithms for behaviour trees etc. [21].

Another example of an agent playing a commercial RTS game is Berkeley Overmind [4], which plays Starcraft PC game. It has been so successful that it has won the AIIDE 2010 Starcraft competition and even managed to defeat some professional players.

Other related works are various bridges to other computer games like Environment Interface Standard (EIS) [2]. EIS is an interface for connecting of agent platforms to environments, much like Pogamut is. Its goals are mainly identical to those of Pogamut, but means slightly different. For example, their definition of percepts and actions uses ontology editors, while Pogamut uses XML files.

10.2 Future Work

The main goals for future work would be to extend the support for Jason by optimizing the code as much as possible and also to overcome the bugs of Defcon AI API mentioned in 2.6.

We do consider seriously designing a series of tests for the agents, that would examine their ability to act with broken communication lines or a rouge fleet breaking off to provoke an ally into attacking etc.

Finally, we may create an implementation of a much better AI player, with much more advanced behaviour in order to test PogamutDefcon usability even better. Using planners or better evaluation of the current battle is certainly desirable. We have considered planners like GOAP (Goal-Oriented Action Planner) [23], which is a simple planner without negative effects. We may try to predict locations of enemy fleets much better, or try to evolve various aspects of the AI, from best placement locations for buildings and fleets, or learn through machine learning a better composition for fleets for various matchups. We may add more proactive behaviour of fleet agents with higher degree of cooperation, like defensive fleets, which would patrol player's own coasts and protect the land against enemy units like sneaking submarines.

Conclusion

We have created a reasonable bridge between Pogamut and Defcon AI API, which allowed us to control a set of agents cooperatively playing Defcon as one of the players. Through the power of Pogamut worldview we have managed to introduce a powerful tool to AI programming for Defcon and we have also included means for a stable access to Defcon AI API querying functions. We have implemented a set of tools for map analysis simplifying this difficult task for the user. From a simple information about each location of a sampling grid, we have constructed a representation of the area borders using quad trees and finally transform it to a workable fleet and building placement information. We have included means of comfortable agent launching and automated the whole project building process to keep it as simple for the user as possible. We have shown how to implement a MAS-styled group of agents playing Defcon on par with the internal Defcon AI. We have provided a small framework connecting PogamutDefCon to Jason BDI engine. Furthermore we have shown how to use it and implement a group of agents with their main reasoning being placed inside Jason. Finally we have evaluated both of these groups of agents and provided results.

We therefore conclude, that we have reached the goals for this thesis as stated on page 6.

Bibliography

[1] BAUMGARTEN, Robin. *Combining Artificial Intelligence Methods: Automating the Playing of DEFCON*. 2007. Master Thesis. Imperial College London.

[2] BEHRENS, Tristan; HINDRIKS, Koen; DIX, Jürgen. Towards an environment interface standard for agent platforms. In *Annals of Mathematics and Artificial Intelligence*. November 2010, p. 1-35.

[3] BELLIFEMINE, Fabio; POGGI, Agostino; RIMASSA, Giovanni. JADE - A FIPA-compliant agent framework. In *Proceedings of PAAM'99*, London, 1999, p. 97-108.

[4] *Berkeley Overmind* [online]. [cit. 2011-08-28]. URL <`http://overmind.cs.berkeley.edu/`>.

[5] BOER Frank S. de; HINDRIKS Koen V.; HOEK Wiebe van der; MEYER John-Jules Ch. Agent Programming with Declarative Goals. In *Intelligent Agents VII Agent Theories Architectures and Languages 7th International Workshop*, LNCS 1986, Springer, 2000, p. 228-243.

[6] *BotPrize* [online]. [cit. 2011-08-28]. URL <`http://botprize.org/`>.

[7] BORDINI, Rafael H.; HÜBNER, Jomi F. An overview of Jason. In *Association for Logic Programming Newsletter*. 2006, vol. 19, no. 3.

[8] BRATMAN, Michael E. *Intentions, Plans, and Practical Reason*. Harvard University Press, 1999. ISBN 9781575861920.

[9] BRYSON, Joanna J. *Intelligence by design: Principles of Modularity and Coordination for Engineering Complex Adaptive Agent*. 2001. PhD Thesis. MIT, Department of EECS, Cambridge, MA.

[10] DASTANI, Mehdi. Modular Rule-Based Programming in 2APL. In GIURCA, Adrian; GASEVIC, Dragan; TAVETER, Kuldar. *Handbook of Research on Emerging Rule-Based Languages and Technologies: Open Solutions and Approaches*. 2 Volumes. 2009. ISBN 978-1-60566-402-6.

[11] *Defcon* [online]. Introversion software Limited. [cit. 2011-08-20]. URL <`http://www.introversion.co.uk/defcon/`>.

[12] *Defcon AI API* [online]. [cit. 2011-08-20]. URL <`http://www.doc.ic.ac.uk/~rb1006/projects:api`>.

[13] *Eclipse* [online]. The Eclipse Foundation, Inc. [cit. 2011-08-21]. URL <`http://www.eclipse.org/`>.

[14] GEMROT, Jakub. *Joint Behaviour for Virtual Humans*. 2007. Master Thesis. Charles University in Prague.

[15] GEMROT, Jakub; KADLEC, Rudolf; BIDA, Michal; BURKERT, Ondrej; PIBIL, Radek; HAVLICEK, Jan; ZEMCAK, Lukas; SIMLOVIC, Juraj; VANSA, Radim; STOLBA, Michal; PLCH, Tomas; BROM, Cyril. Pogamut 3 Can Assist Developers in Building AI (Not Only) for Their Videogame Agents. In *Agents for Games and Simulations*, LNCS 5920, Springer, 2009, p. 1-15.

[16] *Guice* [online]. Google Inc. [cit. 2011-08-21].
URL <`http://code.google.com/p/google-guice/`>.

[17] HINDRIKS, Koen V., et al. Formal Semantics for an Abstract Agent Programming Language. In SINGH, Munindar P.; RAO, Anand; WOOLDRIDGE, Michael J. *Intelligent Agents IV (LNAI 1365)*. 1998. p. 215-229.

[18] HÜBNER, Jomi; SICHMAN, Jaime; BOISSIER Olivier. MOISE+: towards a structural, functional, and deontic model for MAS organization. In *Proceedings of the first international joint conference on Autonomous agents and multiagent systems: part 1*. Bologna, Italy : ACM, 2002. p. 501-502. ISBN 1-58113-480-0.

[19] KADLEC, Rudolf. *Evolution of intelligent agent behaviour in computer games*. 2008. Master Thesis. Charles University in Prague.

[20] KAMINKA, Gal A.; VELOSO, Manuela M.; SCHAFFER, Steve; SOLLITTO, Chris; ADOBBATI, Rogelio; MARSHALL, Andrew N.; SCHOLER, Andrew; TEJADA, Sheila. GameBots: A Flexible Test Bed for Multiagent Team Research. In *Communications of the ACM*. 2002, vol. 45, no. 1, p. 43–45.

[21] LIM, Chong-U; BAUMGARTEN, Robin; COLTON, Simon. Evolving Behaviour Trees for the Commercial Game DEFCON. In *Proceedings of EvoStar : EvoGAMES track*. Springer, 2010. p. 100-110.

[22] *NetBeans* [online]. Oracle Corporation. [cit. 2011-08-20].
URL <`http://netbeans.org/`>.

[23] ORKIN, Jeff. Symbolic Representation of Game World State: Toward Real-Time Planning in Games. In *Proceedings of the AAAI Workshop on Challenges in Game AI*. 2004.

[24] *Pogamut* [online]. AMIS group at Charles University. [cit. 2011-08-20].
URL <`http://www.pogamut.cuni.cz`>.

[25] *Project Emohawk* [online]. AMIS Group at Charles University. [cit. 2011-08-21].
URL <`http://artemis.ms.mff.cuni.cz/emohawk/`>.

[26] RAO Anand S. AgentSpeak(L): BDI Agents speak out in a logical computable language In *Proceedings of MAAMAW*. 1996. p. 42-55.

[27] RAO Anand S. Decision Procedures for Propositional Linear-Time Belief-Desire-Intention Logics In *Lecture Notes in Computer Science*. 1996, vol. 1037, p. 33-48.

[28] RUSSEL, Stuart J.; NORVIG, Peter. *Artificial Intelligence: A Modern Approach*. 2nd ed. Upper Saddle River, New Jersey: Prentice Hall, 2003. ISBN 0-13-790395-2.

[29] *Starcraft* [online]. Blizzard Entertainment, Inc. [cit. 2011-08-20]. URL <`http://us.blizzard.com/en-us/games/sc/`>.

[30] *Unreal Development Kit* [online]. Epic Games, Inc. [cit. 2011-08-20]. URL <`http://www.udk.com/`>.

[31] *Unreal Runtime 2* [online]. Epic Games, Inc. [cit. 2011-08-20]. URL <`http://apacudn.epicgames.com/Two/UnrealEngine2Runtime22262002.html`>.

[32] *Unreal Tournament 3* [online]. Epic Games, Inc. [cit. 2011-08-20]. URL <`http://www.unrealtournament.com/`>.

[33] WOOLDRIDGE, Michael. *Reasoning about Rational Agents*. 1st ed. The MIT Press. 2000. ISBN 0262232138.

List of Tables

Java AI implementation's game results 9.1 on page 47.
Jason AI implementation's game results 9.2 on page 48.

List of Used Acronyms

AI - Artificial Intelligence
2APL - A Practical Agent Programming Language
3APL - An Abstract Agent Programming Language
API - Application programming interface.
AOP - Agent-oriented programming
BDI - Belief-Desire-Intention
BDICTL - Belief-Desire-Intention Computational Tree Logic
CTL - Computational Tree Logic
DLL - Dynamic-Link Library
EIS - Environment Interface Standard
FPS - First-Person Shooter
FSM - Finite State Machine
GOAP - Goal-Oriented Action Planner
IDE - Integrated Development Environment
JADE - Java Agent DEvelopment Framework
JDBC - Java DataBase Connectivity
JNI - Java Native Interface
JVM - Java Virtual Machine
LORA - Logic of Rational Agents
MAS - Multi-Agent System
OOP - Object-Oriented Programming
POSH - Parallel-rooted Ordered Slip-stack Hierarchical
RTS - Real-Time Strategy
SACI - Simple Agent Communication Infrastructure
UDK - Unreal Development Kit
UR2 - Unreal Runtime 2
UT - Unreal Tournament
UT2004 - Unreal Tournament 2004
XML - eXtensible Markup Language